

TURING 图灵程序设计丛书

日本C语言入门第一书

原版畅销**20万册**



明解 C语言

「日」柴田望洋 著
管杰 罗勇 译

 **人民邮电出版社**
POSTS & TELECOM PRESS

TURING

图灵程序设计丛书

明解 语言

〔日〕柴田望洋 著
管杰 罗勇 译

人民邮电出版社
北京

图书在版编目 (CIP) 数据

明解 C 语言 / (日) 柴田望洋著 ; 管杰, 罗勇译.
—北京 : 人民邮电出版社, 2013.5
(图灵程序设计丛书)
ISBN 978-7-115-29979-6

I . ①明… II . ①柴… ②管… ③罗… III . ①
C 语言—程序设计—教材 IV . ① TP312

中国版本图书馆 CIP 数据核字 (2012) 第 267752 号

内 容 提 要

本书是日本的 C 语言经典教材, 自出版以来不断重印、修订, 被誉为“C 语言圣经”。作者在日本 IT 界家喻户晓, 出版过一系列极富影响力的计算机教材和参考书。其简洁、通俗的文风深受读者的喜爱。

本书图文并茂, 示例丰富, 设有 190 段代码和 164 幅图表, 对 C 语言的基础知识进行了彻底剖析, 内容涉及数组、函数、指针、文件操作等。对于 C 语言语法以及一些难以理解的概念, 均以精心绘制的示意图, 清晰、通俗地进行讲解。

本书适合 C 语言初学者阅读。

图灵程序设计丛书

明解 C 语言

-
- ◆ 著 [日] 柴田望洋
 - 译 管 杰 罗 勇
 - 责任编辑 傅志红
 - 执行编辑 乐 馨 张 靖
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
 - 邮编 100061 电子邮件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 北京天宇星印刷厂印刷
 - ◆ 开本: 800×1000 1/16
 - 印张: 21.75
 - 字数: 514 千字 2013 年 5 月第 1 版
 - 印数: 1-5 000 册 2013 年 5 月北京第 1 次印刷
 - 著作权合同登记号 图字: 01-2012-3527 号
 - ISBN 978-7-115-29979-6
-

定价: 59.00 元

读者服务热线: (010)51095186 转 604 印装质量热线: (010)67129223

反盗版热线: (010)67171154

目录

Content

第1章 初识C语言

1-1 显示计算结果

计算整数的和并显示结果	2
程序和编译	2
注释	3
固定代码	4
格式化输出函数printf	4
语句	5
计算并显示整数的差	5
格式化字符串和转换说明	6
符号的称呼	7
无格式化输出	8
字符串常量	9

转义字符	9
------------	---

1-2 变量

常量和变量	10
声明多个变量	11
赋值	11

1-3 输入和显示

通过键盘进行输入	12
格式化输入函数scanf	12
乘法运算	13
输出函数puts	14

第2章 运算和数据类型

2-1 运算

四则运算	18
运算符和操作数	18
商和余数	19
乘除运算符和加减运算符	19
使用printf函数打印 %	19
获取整数的最后一位数字	20
多个转换说明	21
单目运算符	22
赋值运算符	23
表达式和赋值表达式	23

表达式语句	23
-------------	----

2-2 数据类型

求平均值	24
数据类型	24
int类型和double类型	25
数据类型和对象	26
整数常量和浮点数常量	27
double类型的运算	27
数据类型和运算	28
类型转换	30
转换说明	32

第3章 分支结构程序

3-1 if 语句

程序的流程	36
if 语句 (1)	36
奇数的判定	37
if 语句 (2)	38
奇数和偶数	39
判断	39
非0的判断	40
语法结构	40
相等运算符	42
比较余数	43
关系运算符	44

嵌套的if语句	45
计算较大值	46
计算三个数的最大值	47
条件运算符	48
差值计算	49
复合语句 (程序块)	50
判断季节	52
逻辑运算符	52

3-2 switch语句

程序的流程	54
switch语句和if语句	57
选择语句	57

第4章 程序的循环控制

4-1 do语句

do语句	60
复合语句 (程序块) 中的声明	61
逻辑非运算符	61
逆向显示整数值	62
计算整数的位数	63
初始化	64
复合赋值运算符	66
后置递增运算符和 后置递减运算符	67

4-2 while语句

while语句	68
字符常量	69
putchar	69

用递减运算符简化程序代码	70
数据递增	71
限定次数的循环操作	72
前置递增运算符和 前置递减运算符	73

4-3 for语句

for语句	74
循环语句	75
固定次数的循环	76

4-4 多重循环

九九乘法表	78
多重循环	79
长方形	80

直角三角形	80
4-5 程序的组成元素和格式	
关键字	82
标识符	82
分隔符	83

运算符	83
常量和字符串常量	83
自由的书写格式	84
连接相邻的字符串常量	85
缩进	85

第5章 数组

5-1 数组	
数组	88
数组和for语句	90
数组初始化	92
数组的复制	93
输入数组元素的值	94
对数组进行倒序排列	94
使用数组进行成绩处理	96
对象式宏	96
赋值表达式	98
及格学生一览表	100
数组的元素个数	100
成绩分布图	101

5-2 多维数组	
矩阵	102
5-3 质数计算	
质数	104
质数计算程序（第1版）	104
大整数	104
break语句	104
质数计算程序（第2版）	106
质数计算程序（第3版）	107
质数计算程序（第4版）	108
质数计算程序（第5版）	110
逗号运算符	110

第6章 函数

6-1 什么是函数	
main函数	114
库函数	114
函数定义和函数调用	114

三个数中的最大值	118
平方差	119
幂	120
值传递	120

调用其他函数	121
6-2 函数设计	
没有返回值的函数	122
通用性	122
不含形参的函数	124
函数返回值的初始化	125
作用域	125
计算最高分的程序	126
文件作用域	126
声明和定义	127
函数原型声明	127
头文件和文件包含指令	128

函数的通用性	129
数组的传递	130
对接收到的数组进行写入处理	132
const类型的修饰符	133
顺序查找	134
哨兵查找法	135
表达式语句和空语句	137
多维数组的传递	138

6-3 作用域和存储期	
作用域和标识符的可见性	140
存储期	142

第7章 基本数据类型

7-1 基本数据类型和数	
基本数据类型	148
基数	149
基数转换	150
7-2 整型和字符型	
字符型和整型	152
<limits.h>头文件	154
字符型	155
sizeof运算符	156
size_t型和typedef声明	157
整型的灵活运用	157
整型常量	158
整型常量的数据类型	158
内部表示和位	160
无符号整数的内部表示	160
有符号整数的内部表示	161
反码表示法和补码表示法	162
窥探整数内部	163
按位操作的逻辑运算	164

位移运算符	166
位数的计算	168
求出unsigned型的位数	168
显示位的内容	169
整数的显示	170
数据溢出和异常	171

7-3 浮点型	
浮点型	172
浮点型常量	173
循环的控制	174
<math.h>头文件	175

7-4 运算	
运算符一览	176
优先级	176
结合性	176
数据类型转换	178
sizeof运算符	180
sizeof运算符和数组	181

第8章 动手编写各种程序吧

8-1	函数式宏	
	函数和数据类型	184
	函数式宏	185
	函数和函数式宏	186
	不带参数的函数式宏	187
	函数式宏和逗号运算符	188
8-2	枚举类型	
	枚举类型	190
	枚举常量	192
	命名空间	193
8-3	递归	
	阶乘	194

	最大公约数	196
	问题和递归	197
8-4	输入输出和字符	
	数字字符计数	198
	getchar函数	198
	EOF	198
	字符和数值	199
	字符	200
	转义字符	203
	复制	204

第9章 字符串的基本知识

9-1	什么是字符串	
	字符串字面量	208
	字符串字面量的长度	208
	字符串	210
	字符数组的初始化赋值	211
	空字符串	212
	字符串的读取	212
	格式化显示字符串	213
9-2	字符串数组	
	字符串数组	214

	读取字符串数组中的字符串	215
9-3	字符串处理	
	字符串长度	216
	遍历字符串	218
	数字字符的出现次数	219
	字符串数组的参数传递	220
	大小写字符转换	222

第10章 指针

10-1 指针

函数的参数	226
变量和对象	227
地址	227
取址运算符	228
指针	229
指针运算符	231

10-2 指针和函数

作为函数参数的指针	232
二值互换	234

引用传递 (C++语言)	235
计算和与差	236
scanf函数和指针	236
将两个值升序排列	237
指针的类型	238
标量型	239

10-3 指针和数组

指针和数组	240
数组的传递	244

第11章 字符串和指针

11-1 字符串和指针

字符串和指针	248
数组和指针的相同点	249
数组和指针的不同点	250
字符串数组	252

11-2 通过指针操作字符串

字符串和指针	254
判断字符串长度	254

const	254
使用指针进行遍历	255
字符串的复制	256
不正确的字符串复制	258
返回指针的函数	259

11-3 字符串处理库函数

字符串处理函数	260
字符串转换函数	264

第12章 结构体

12-1 结构体

排序	268
冒泡排序法	269
数据关联性	270
结构体	272
结构体成员（.运算符）	274
成员的初始化	275
结构体成员（->运算符）	276
结构体和typedef	278
结构体和程序	279

聚合类型	280
命名空间	280
返回结构体的函数	281
结构体数组	282
派生类型	282
表示日期和时间的结构体	284

12-2 作为成员的结构体

表示坐标的结构体	286
表示具有定位功能的汽车的 结构体	286

13-1 文件与流

文件	290
流	290
标准流	291
FILE型	291
打开文件	292
关闭文件	294
打开与关闭文件示例	295
文件数据汇总	296
写入日期和时间	298

第13章 文件处理

获取上一次运行时的信息	300
标准输入输出	302
显示文件内容	302
文件的复制	304

13-2 文本和二进制

在文本文件中保存实数	306
文本文件和二进制文件	307
在二进制文件中保存实数	308
显示文件自身	310

附录 1 C语言简介

C语言的历史.....	314
K&R——C语言的圣经	314

C语言标准规范.....	314
--------------	-----

附录 2 printf函数与scanf函数

printf函数	318
----------------	-----

scanf函数	322
---------------	-----

致谢	326
----------	-----

参考文献	326
------------	-----

索引	327
----------	-----

版权声明	338
------------	-----

第 1 章

初识C语言

如果说熟悉了一件事就能大幅进步的话，那么长期从事这件事并已完全熟练的人就应该是高手了。但现实并非如此，就拿体育训练来说，假如训练的方式是错误的，只会越练习越差。编程也是如此，仅仅熟练是不够的。

总之，任何事情开始的时候，都需要先试试水。本章就带领大家尝试一下简单的 C 语言编程。



1-1 显示计算结果

计算整数的和并显示结果

电脑也称为电子计算机，对它来说，任何任务都是通过计算来完成的。那么就让我们使用C语言来进行下面的计算吧。

计算整数 15 和 37 的和，并显示结果。

在编辑器中键入代码清单 1-1 所示的程序代码。C 语言程序是区分大小写和全半角字符的，请大家在书写的时候特别注意。

代码清单 1-1

```
/*  
    显示整数 15 和 37 的和  
*/  
  
#include <stdio.h>  
  
int main(void)  
{  
    printf("%d", 15 + 37);    /* 用十进制数显示整数 15 和 37 的和 */  
  
    return (0);  
}
```

运行结果

52

► 为了方便读者阅读，本书中的代码使用了蓝色、斜体、consolas 体和 Courier New 体来进行区分。大家在书写程序代码的时候可以忽略这些字体设置。

程序和编译

如代码清单 1-1 所示，人们通过字符序列创建出的程序被称为**源程序**（source program），用来保存源程序的文件被称为**源文件**（source file）。

习惯上我们把 C 语言源文件的扩展名约定为“.c”，例如我们可以把源文件保存为 list0101.c。

通过字符序列创建出的程序，需要转换为计算机能够理解的位序列，也就是 0 和 1 的序列。源程序通常需要进行如图 1-1 所示的翻译操作之后才能执行（关于位的介绍请参考第 7 章）。

完成这些翻译工作之后运行程序，屏幕上就能显示出结果 52 了。

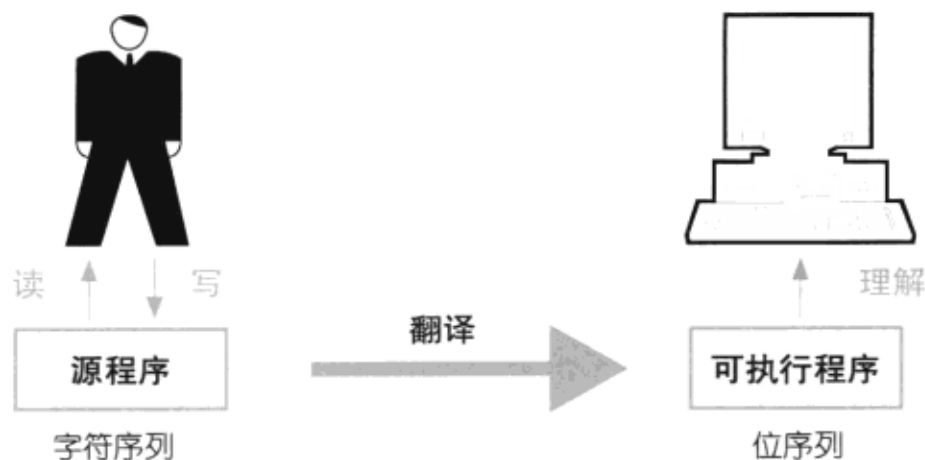


图 1-1 源程序和可执行程序

► 编译器和运行环境不同时，翻译的步骤和程序的执行方法也不同，请大家参考各自编译器的说明书。在后面的专题 1-1 中会对翻译和编译器等术语进行说明。

源程序中如果有拼写错误，翻译的时候就会发生错误，并显示出相应的**诊断消息**。出现这种情况时请仔细检查键入的程序代码，纠正错误之后再进行编译。

程序中有着大量 # 和 { 等符号，大家可能不理解它们的意思。不过没关系，我们慢慢来，一点一点学习。

► 稍后我们还会对符号的称呼进行总结。

注释

源程序中 /* 和 */ 之间的部分，称为**注释** (comment)^①。有没有注释以及注释的内容如何，其实对程序的运行并没有什么影响。编程者用简洁明了的语言将程序想要表达的意思标注在程序旁，这样能提高程序的可读性。

► 代码清单 1-1 这样简短的程序还好，如果碰到他人编写的长篇程序，理解起来就非常困难了。另外，即使是自己编写的程序也不可能永远都记得住。因此，适当加一些注释不仅能方便他人，也能方便编程者自己。

■ 注意 ■

请大家在源程序中，用简洁的语言把想要表达的意思以**注释**的形式记录下来。

从程序中可以看出，注释也可以是多行的。但是请大家注意不要把结束注释用的符号误写成 “/*”，否则后面的程序都会被解释为注释。

^① C99支持单行注释，即 “//……” 这种形式，“//” 之后直到行尾的内容为注释。（本书脚注均为译者注）

固定代码

如图1-2所示，删除程序中的注释。白底以外的部分是一段固定代码，它的含义之后会详细介绍，请大家牢记这段代码。

现阶段我们暂时先照搬这段代码，其余的部分由自己编写。

```
#include <stdio.h>

int main(void)
{
    printf("%d", 15 + 37);

    return (0);
}
```

图 1-2 程序和固定代码

格式化输出函数 printf

printf 函数可以在显示器上进行输出操作(**printf** 末尾的 f 源自 format(格式化)这个单词)。

如果想要使用某个函数的功能，就必须通过**函数调用** (function call) 来实现。调用**printf**函数显示15和37的和的过程如图1-3所示。

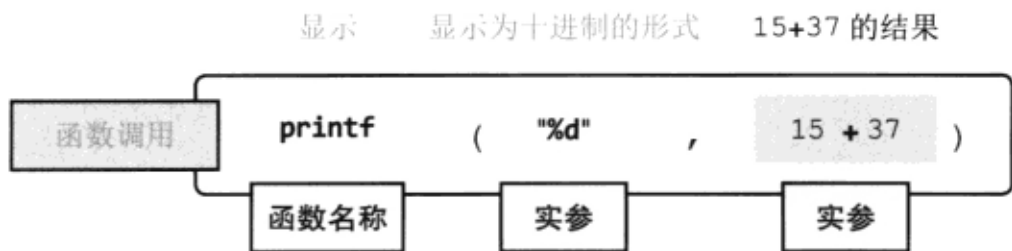


图 1-3 调用 printf 函数

调用此函数即发出了“显示这些内容”的请求，然后通过括号中的**实参** (argument) 来传递想要显示的内容。另外，如本例所示，当实参超过两个时，需要用逗号隔开。

printf 函数的第一个实参 **"%d"** 指定了输出格式，它告诉程序：以十进制数的形式显示后面的实参。因此，通过调用 **printf** 函数显示出了第二个实参 15+37 的值，即 15 与 37 的和 52。

► **"%d"** 中的 d 源自 decimal (十进制数)。十进制和二进制将会在 7-1 节进行介绍，八进制数和十六进制将会在 7-2 节进行介绍。**printf** 函数的详情，请参考附录 2 中的总结。

■ 注意 ■

函数调用是申请进行处理请求，而调用函数时的一些辅助指示则通过**实参**来发出。

语句

请大家仔细观察之前的程序代码，调用 **printf** 函数的时候使用了分号，那段固定代码中也使用了分号（**return**(0);）。这里的分号就相当于中文里的句号。

正如在句子末尾加上句号才能构成完整的一句话，C 语言也需要在末尾加上分号来构成正确的**语句**（statement）。

■ 注意 ■

原则上语句必须以分号结尾。

计算并显示整数的差

代码清单 1-2 所示程序的功能是计算并显示 15 减去 37 的差。

代码清单 1-2

```
/*
 * 计算并显示 15 减去 37 的差
 */

#include <stdio.h>

int main(void)
{
    printf("%d", 15 - 37);    /* 用十进制数显示 15 减去 37 的值 */

    return (0);
}
```

运行结果

-22

运行程序就会显示结果 -22。可以看到当计算结果为负数时，会自动加上负号。

专题 1-1 翻译阶段和编译

运行 C 语言之前，理论上要经过 8 个**翻译阶段**（translation phase）。另外，运行源代码还需要安装必要的软件环境，也就是**编译器**^①。

大多数 C 语言编译器都是通过**编译方式**（如本文中描述的方式）把源代码翻译成计算机能够直接理解执行的形式。但是也存在逐行解释然后执行的**解释方式**（执行速度比较缓慢）。

^① 即符合 C 语言规范的实现（implementation）。

格式化字符串和转换说明

程序运行的时候如果只显示和或者差的值，理解上会比较困难，接下来我们让结果显示得更加人性化一些，请看代码清单 1-3 所示程序。

这次我们把 **printf** 函数的第一个实参设置得更长更复杂一些。

代码清单 1-3

```
/*
    人性化地显示 15 与 37 的和
*/

#include <stdio.h>

int main(void)
{
    printf("15 与 37 的和是 %d。 \n", 15 + 37);    /* 显示结果后换行 */

    return (0);
}
```

运行结果

15 与 37 的和是 52。

清单中的蓝色底纹部分是 **printf** 函数的第一个实参，被称为**格式化字符串**（format string）。

格式化字符串中的 **%d** 指定了实参要以十进制数的形式显示，这就是**转换说明**（conversion specification）。格式化字符串中没有指定转换说明的字符基本上都会原样输出。

格式化字符串结尾的 **\n** 是代表**换行**（new line）的符号，**** 和 **n** 组成了一个特殊的“换行符”。

► 在字符串结尾加上换行符的必要性请参考专题 1-2。

本程序的显示结果如图 1-4 所示。

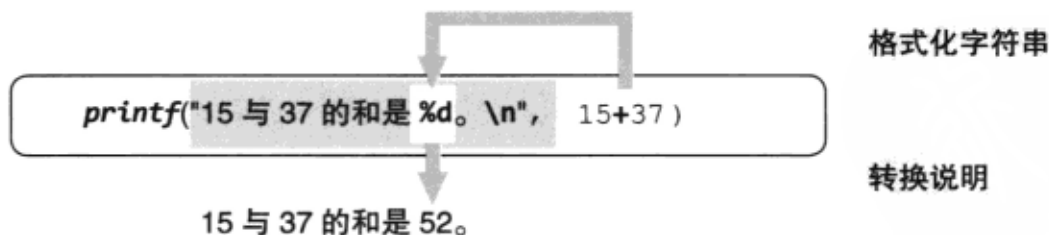


图 1-4 格式化字符串和转换说明

● 练习 1-1

编写一段程序，计算出 15 减去 37 的结果，并以“15 减去 37 的结果是 -22。”的格式进行显示。

符号的称呼

C 语言里符号的称呼如表 1-1 所示。

■ 表 1-1 符号的称呼

+	加号、正号、加	{	左大括号
-	减号、负号、连字符、减	}	右大括号
*	星号、乘号、米号、星	[左方括号、左中括号
/	斜线、除号]	右方括号、右中括号
\	反斜线	<	小于
¥	货币符号	>	大于
%	百分号	?	问号
.	点	!	感叹号
,	逗号	&	and 符
:	冒号	~	波浪线
;	分号	^	音调符号
'	单引号	#	井号
"	双引号	_	下划线
(左括号、左圆括号、左小括号	=	等号、等于
)	右括号、右圆括号、右小括号		竖线

专题 1-2 换行的必要性

假设代码清单 1-1 的执行程序名是 `list0101`。在大多数运行环境中的情况如下方左图所示，程序的输出结果 52 后面直接带着提示符（▷是操作系统的提示符，例如 MS-DOS 的提示符是 `A>`，UNIX 的提示符是 `%` 等）。因此，像代码清单 1-3 的显示结果是：15 与 37 的和是 52。在程序最后加上换行符更好一些（如下方右图所示）。

```
▷list0101□
52▷
```

```
▷list0103□
15与37的和是52。
▷
```


无格式化输出

调用 **printf** 函数的时候也可以只使用一个参数。这时，格式化字符串内的字符将按照原样显示。显示“您好！我叫柴田望洋。”的程序如代码清单 1-4 所示。

代码清单 1-4

```
/*
 打招呼并进行自我介绍
*/

#include <stdio.h>

int main(void)
{
    printf("您好！我叫柴田望洋。\\n");

    return (0);
}
```

运行结果

您好！我叫柴田望洋。

下面我们稍微把程序修改一下，让“您好！”和“我叫柴田望洋。”分别在两行显示。修改后的程序如代码清单 1-5 所示。

代码清单 1-5

```
/*
 打招呼并进行自我介绍（打招呼 and 自我介绍分行显示）
*/

#include <stdio.h>

int main(void)
{
    printf("您好！\\n我叫柴田望洋。\\n"); /* 中间换行 */

    return (0);
}
```

运行结果

您好！
我叫柴田望洋。

在格式化字符串中间插入 **\\n** 就可以实现换行操作。而像代码清单 1-6 那样，调用两次 **printf** 函数也可以得到同样的效果。

代码清单 1-6

```
/*
 打招呼并进行自我介绍（打招呼 and 自我介绍分行显示）
*/

#include <stdio.h>

int main(void)
{
    printf("您好！\\n"); /* 显示后换行 */
    printf("我叫柴田望洋。\\n"); /* 显示后换行 */

    return (0);
}
```

运行结果

您好！
我叫柴田望洋。

字符串常量

像“ABC”和“您好!”这样用双引号括起来的一连串连续排列的文字,被称为**字符串常量**(string literal)。

► 原本在字符串常量中使用汉字等全角文字是违反规定的。但在日本使用的编译器大部分都已经支持全角文字了,所以为了让读者更容易理解,本书中也使用了全角文字。

转义字符

我们已经介绍了能够实现换行的特殊符号 `\n`,像这样的特殊符号被称为**转义字符**(escape sequence)。

响铃(alert)的转义字符是 `\a`。代码清单 1-7 中的程序,在显示“您好!”之后响铃 3 次。

代码清单 1-7

```
/*
 * 响铃三次
 */
#include <stdio.h>

int main(void)
{
    printf("您好! \a\a\a\n"); /* 在显示的同时发出三次响铃 */
    return (0);
}
```

运行结果

您好!

► 程序在某些环境下运行,可能不响铃(通常情况下都是发出蜂鸣音,即“哔”的声音,但有时并不发出声音,而是通过视觉来发出警报)或者连续响铃三次。

● 练习 1-2

编写一段程序,调用一次 `printf` 函数,显示右侧内容。

风
林
火
山

● 练习 1-3

编写一段程序,调用一次 `printf` 函数,显示右侧内容。

喂!
您好!

再见。

1-2 变量

常量和变量

到目前为止，我们都是对传入程序中的**常量**（constant）进行求和、求差等操作，并显示出计算结果。如果遇到比较复杂的计算，为了在中途记录结果就需要使用**变量**了。

听到“变量”这个词，不喜欢数学的人可能会联想到上中学时学到的方程式，产生畏难情绪。其实并不用担心，请看下文。

■ 注意 ■

变量其实就是用来放置数值等内容的“盒子”。

想要使用这个可以存放数值等内容的魔法盒，就必须遵循一定的流程。首先需要提前进行如下**声明**（declaration）。

```
int vx;
```

如图 1-5 所示，我们通过声明就准备出了一个名为 vx 的盒子。

这个盒子只能用来存放整数值，因此 vx 被称为**整型**（int 型）。

► int 是整数的英文单词 integer 的缩写。



图 1-5 声明变量

虽然本例中使用的变量名是 vx，但其实变量可以自由命名，像 i 或者 x 这样只有一个字符的变量名也是可以的。

► 变量的命名规则请参考 4-5 节。

让我们考虑下面这个问题，使用变量编写一段程序吧。

给变量赋一个合适的值并显示。

程序如代码清单 1-8 所示。

代码清单 1-8

```
/*
 为两个变量赋整数值并显示
*/

#include <stdio.h>

int main(void)
{
    int vx, vy;                /* vx 和 vy 是 int 类型的变量 */

    vx = 57;                   /* 把 57 赋给 vx */
    vy = vx + 10;              /* 把 vx+10 赋给 vy */

    printf("vx 的值是 %d。\\n", vx); /* 显示 vx 的值 */
    printf("vy 的值是 %d。\\n", vy); /* 显示 vy 的值 */

    return (0);
}
```

运行结果

vx 的值是 57。

vy 的值是 67。

声明多个变量

上述程序通过逗号分隔，在一行中声明了两个变量 vx 和 vy，这样就创建了名为 vx 的变量和名为 vy 的变量。

当然也可以像右边这样分行声明两个变量。

- ▶ 分行声明变量更便于添加注释，并且也能更容易地添加和删除声明，但是程序的代码行数会有所增加。所以请大家根据实际情况灵活使用这两种声明方式。

```
int vx;
int vy;
```

另外，本程序在声明之后并未书写任何内容，而是空出一行，这样增加了程序的可读性。

赋值

在本程序中我们第一次使用了等号 =，它表示把右侧的值赋给左侧的变量。因此，首先会把 57 赋给变量 vx（图 1-6）。

另外，任何时候都可以取出变量的值。我们取出了 vx 的值并加上 10，然后再赋给 vy，这样 vy 的值就变成了 67。

当然，大家一定不要忘记在语句的末尾加上分号。

```
vx = 57;
vy = vx + 10;
```

- ▶ 需要注意，这里的等号并不像数学中那样代表 vx 和 57 相等之意。

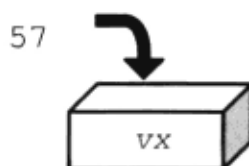


图 1-6 为变量赋值

1-3 输入和显示

通过键盘进行输入

仅仅输出显示没有什么意思，下面我们来读取通过键盘输入的值，模拟人机对话。

读取一个整数值，并显示出来进行确认。

程序如代码清单 1-9 所示。

代码清单 1-9

```
/*
    显示并确认输入的整数值
*/

#include <stdio.h>

int main(void)
{
    int no;

    printf("请输入一个整数:");
    scanf("%d", &no);          /* 读取整数的值 */

    printf("您输入的是 %d。\\n", no);

    return (0);
}
```

运行结果

请输入一个整数: 37
您输入的是 37。

► 为了进行区别，用 `consolas` 体显示运行结果中输入的整数 37。如果输入的整数是 53，就会显示：“您输入的是 53。”

格式化输入函数 `scanf`

`scanf` 函数可以从键盘读取输入的信息。

```
scanf("%d", &no);
```

这里同样可以像 `printf` 函数那样，通过转换说明 `"%d"` 来限制函数只能读取十进制数。

因此，上述程序就向计算机传达了这样一个指令：从键盘读取输入的十进制数，并把它保存到 `no` 中（图 1-7）。

与 `printf` 函数不同的是，这里的变量名之前必须加上一个特殊符号 `&`，请大家注意。

► `&` 的具体含义会在第 10 章进行说明。

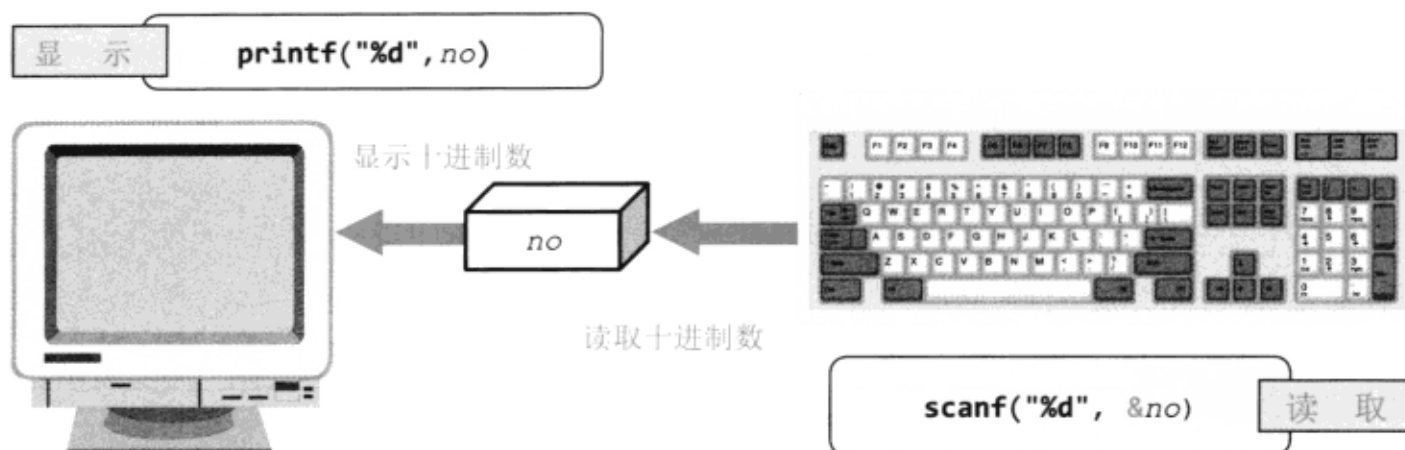


图 1-7 输出函数 printf 和输入函数 scanf

乘法运算

读取一个整数并显示其 10 倍数的值（代码清单 1-10）。

代码清单 1-10

```
/*
 * 读取一个整数并显示其 10 倍数的值
 */
#include <stdio.h>

int main(void)
{
    int no;

    printf("请输入一个整数:");
    scanf("%d", &no);

    printf("它的 10 倍数是 %d。\\n", 10 * no);

    return (0);
}
```

运行结果

请输入一个整数: 357
它的 10 倍数是 3570。

大家在这里第一次接触到了符号 `*`，它是乘法运算的运算符。请大家注意，乘法运算符不是 `×` 而是 `*`。当然程序中的“`10*no`”也可以写作“`no*10`”。

● 练习 1-4

编写一段程序，像右面那样读取一个整数并显示该整数加上 10 之后的结果。

请输入一个整数: 57
该整数加上 10 的结果是 67。

● 练习 1-5

编写一段程序，像右面那样读取一个整数并显示该整数减去 10 之后的结果。

请输入一个整数: 57
该整数减去 10 的结果是 47。

输出函数 puts

接下来让我们利用变量来解决稍微复杂一些的问题。

读取两个整数的值，显示它们的和。

程序如代码清单 1-11 所示。

代码清单 1-11

```
/*
 * 显示出读取到的两个整数的和
 */

#include <stdio.h>

int main(void)
{
    int n1, n2;

    puts("请输入两个整数。");
    printf(" 整数 1 : ");    scanf("%d", &n1);
    printf(" 整数 2 : ");    scanf("%d", &n2);

    printf(" 它们的和是 %d。\\n", n1 + n2); /* 显示和 */

    return (0);
}
```

运行结果

请输入两个整数。
 整数 1 : 27
 整数 2 : 35
 它们的和是 62。

► 如本例所示，C 语言允许在同一行中书写多条语句（同一条语句也可以分成多行书写）。程序的书写格式将在第 4 章进行说明。

本例中第一次使用到了 **puts** 函数（末尾的 s 取自 string）。

puts 函数可以顺序输出作为实参的字符串，并在结尾换行。也就是说，**puts("...")** 与 **printf("...\n")** 的功能基本相同（如图 1-8 所示）。

在需要换行且不用进行格式化输出的时候，就可以使用 **puts** 函数来代替 **printf** 函数。

► 请大家注意 **puts** 函数的实参只能有一个。

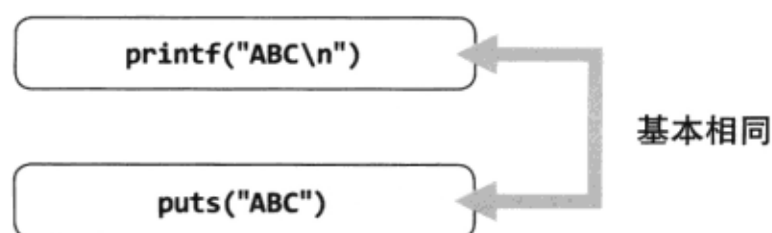


图 1-8 printf 函数和 puts 函数

对本例中的程序进行一些修改（代码清单 1-12），把读取出的整数的和保存在变量 `wa` 中，然后显示出 `wa` 的值。程序的结果和代码清单 1-11 是完全一样的。

代码清单 1-12

```
/*
    显示出读取到的两个整数的和
*/

#include <stdio.h>

int main(void)
{
    int n1, n2;
    int wa;          /* 和 */

    puts("请输入两个整数。");
    printf("整数 1:"); scanf("%d", &n1);
    printf("整数 2:"); scanf("%d", &n2);

    wa = n1 + n2;     /* 把 n1 与 n2 的和赋给变量 wa */

    printf("它们的和是 %d。\\n", wa); /* 显示和 */

    return (0);
}
```

运行结果

请输入两个整数。
整数 1: 27
整数 2: 35
它们的和是 62。

● 练习 1-6

编写一段程序，使用 `puts` 函数显示右侧内容。

风
林
火
山

● 练习 1-7

编写一段程序，像右面这样显示读取到的两个整数的乘积。

请输入两个整数。
整数 1: 27
整数 2: 35
它们的乘积是 945。

● 练习 1-8

编写一段程序，像右面这样显示读取到的三个整数的和。

请输入三个整数。
整数 1: 7
整数 2: 15
整数 3: 23
它们的和是 45。

第 2 章

运算和数据类型

如果有人问你的身高和体重是多少，你会怎么回答呢？也许你会说，我身高 175 厘米，体重 60 公斤。但这些数据都是准确的吗？你的身高正好是 175 厘米吗？即使用身高测量仪测出 175.3 厘米这样的数值，恐怕也是不精确的。你的实际身高可能应该是 175.2869758... 厘米（而且这个数值还会随着时间不断地变化）。但是通常情况下我们说身高 175 厘米，体重 60 公斤就可以了，毕竟不需要精确到那种地步。在程序的世界里也是这样，有时并没有必要表示出精确的实际数值。

本章将会为大家介绍 C 语言进行数值计算时所必备的运算和数据类型等知识。



2-1 运算

四则运算

前一章我们进行了加法、减法和乘法运算，下面我们来尝试除法运算。

读取两个整数的值，然后显示出它们的和、差、积、商和余数。

程序如代码清单 2-1 所示。

代码清单 2-1

```
/*
 * 读取两个整数的值，然后显示出它们的和、差、积、商和余数
 */
#include <stdio.h>

int main(void)
{
    int vx, vy;

    puts("请输入两个整数。");
    printf("整数 vx : "); scanf("%d", &vx);
    printf("整数 vy : "); scanf("%d", &vy);

    printf("vx + vy = %d\n", vx + vy);
    printf("vx - vy = %d\n", vx - vy);
    printf("vx * vy = %d\n", vx * vy);
    printf("vx / vy = %d\n", vx / vy);
    printf("vx %% vy = %d\n", vx % vy);

    return (0);
}
```

运行结果

```
请输入两个整数。
整数 vx : 57
整数 vy : 21
vx + vy = 78
vx - vy = 36
vx * vy = 1197
vx / vy = 2
vx % vy = 15
```

► $vx - vy$ ，只是算出从 vx 中减去 vy 的值，并不是真正的求差运算。也就是说，如果 vy 比 vx 大的话， $vx - vy$ 的值就是负数。求差运算的程序会在第 3 章进行介绍。

运算符和操作数

上一章我们介绍了乘法运算中使用的符号 $*$ ，像这样可以进行运算的符号被称为运算符 (operator)，作为运算对象的变量或者常量被称为操作数 (operand)。例如在加法运算 $vx + vy$ 中， $+$ 就是运算符， vx 和 vy 就是操作数。

运算符左侧的操作数被称为第一操作数或者左操作数，运算符右侧的操作数被称为第二操作数或者右操作数。

► C 语言有很多运算符，7-4 节为大家提供了所有运算符的一览表。

商和余数

求商的运算符是 /。

整数 / 整数

商的整数部分

如上所示，/ 运算只取商的整数部分，也就是说会舍弃小数点以后的部分。例如，5/3 的结果是 1，3/5 的结果是 0。

整数 % 整数

余数

“%”是求余运算。例如，5%3 的结果是 2，3%5 的结果是 3。

► 请大家参照后续的专题 2-1。

乘除运算符和加减运算符

本程序使用的五个运算符，可以大致区分为表 2-1 所示的乘除运算符（multiplicative operator）和表 2-2 所示的加减运算符（additive operator）。

请大家牢记这些运算符的名称。

■ 表 2-1 乘除运算符

双目 * 运算符	$a * b$	a 和 b 的积。
/ 运算符	a / b	a 除以 b 所得到的商（整数之间运算的时候需要舍弃小数点之后的值）。
% 运算符	$a \% b$	a 除以 b 所得到的余数（ a 和 b 都必须是整数）。

■ 表 2-2 加减运算符

双目 + 运算符	$a + b$	a 和 b 的和。
双目 - 运算符	$a - b$	a 减去 b 的值。

► 乘除运算符的英文名称是 binary * operator、/ operator、% operator，加减运算符的英文名称是 binary + operator、binary - operator。

使用 printf 函数打印 %

让我们来看一下输出余数的程序。看上去格式化字符串中的 % 似乎多了一个，但是就像 %d（十进制有符号整数）那样，格式化字符串中的 % 具有转换说明的功能。

```
printf("vx % % vy = %d\n", vx % vy);
```

因此，不使用转换功能，只想输出 % 的时候，必须写成 %%。

► 当使用不具有转换说明功能的 puts 函数来进行输出的时候，就不能写成 %%（这样会输出 %% 的）。

获取整数的最后一位数字

通过灵活地运用求余运算符，我们可以解决下面的问题。

显示读取出的整数的最后一位数字。

程序如代码清单 2-2 所示。

代码清单 2-2

```
/*
 * 显示读取出的整数的最后一位数字
 */
#include <stdio.h>

int main(void)
{
    int no;

    printf("请输入一个整数:");
    scanf("%d", &no);

    printf("最后一位是 %d。\\n", no % 10);

    return (0);
}
```

运行结果 (1)

请输入一个整数: 1357
最后一位是 7。

运行结果 (2)

请输入一个整数: 1780
最后一位是 0。

整数除以 10 的余数就是它的最后一位。

● 练习 2-1

编写一段程序，像右面那样读取两个整数，然后显示出前者是后者的百分之几。

请输入两个整数。

整数A: 54

整数B: 84

A的值是B的64%。

专题 2-1 % 运算符和操作数

利用运算符 % 计算余数，需要遵循如下规则。

■ 如果两个操作数的符号相同，运算的结果就是正数。

正 % 正 → 正

■ 如果两个操作数中至少有一个为负，则运算结果取决于编译器，请大家特别注意这一点。

也就是说，编译器不同，得到的值也有可能不同（因此最好尽量避免这样的运算）。

负 % 负 → 结果取决于编译器

正 % 负 → 结果取决于编译器

负 % 正 → 结果取决于编译器

多个转换说明

读取两个整数，并显示它们的商和余数。程序如代码清单 2-3 所示。

代码清单 2-3

```
/*
 * 读取两个整数，显示它们的商和余数
 */
#include <stdio.h>

int main(void)
{
    int na, nb;

    puts("请输入两个整数。");
    printf("整数 A :"); scanf("%d", &na);
    printf("整数 B :"); scanf("%d", &nb);

    printf("A 除以 B 得 %d 余 %d。 \n", na / nb, na % nb);

    return (0);
}
```

运行结果

请输入两个整数。

整数 A : 57

整数 B : 21

A 除以 B 得 2 余 15。

程序中蓝色底纹部分中包含两个转换说明 `%d`。如图 2-1 所示，这些转换说明分别对应左边数第二个和第三个参数。

需要同时显示两个以上格式化数值时，可以像这样在格式化字符串中使用多个转换说明。

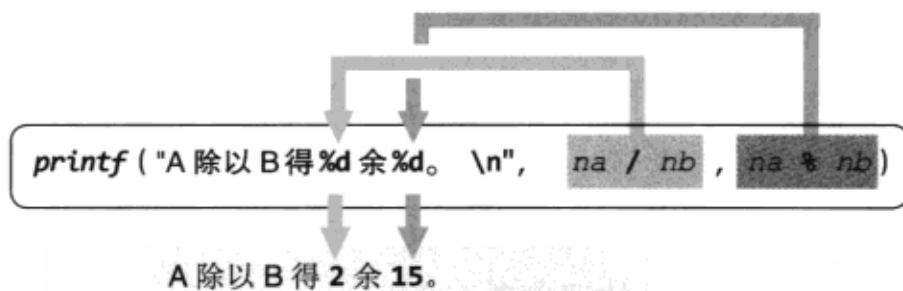


图 2-1 通过 printf 函数显示两个格式化的值

想要一次显示两个以上格式化数值时，可以像这样在格式化字符串中写入多个转换说明。

● 练习 2-2

编写一段程序，像右面那样读取两个整数，然后输出它们的和以及积。

请输入两个整数。

整数A: 54

整数B: 12

它们的和是66，积是648。

单目运算符

我们来考虑一下下面这个问题。

对读取的整数值进行符号取反操作，并输出结果。

也就是输入 75 就显示 -75，输入 -64 就显示 64。程序如代码清单 2-4 所示。

代码清单 2-4

```
/*
 * 对读取的整数值进行符号取反操作，并输出结果
 */

#include <stdio.h>

int main(void)
{
    int num;

    printf("请输入一个整数:");
    scanf("%d", &num);

    printf("符号取反之后的值是 %d.\n", -num); /* 单目运算符 */

    return (0);
}
```

运行结果 (1)

请输入一个整数: 75
符号取反之后的值是
-75。

运行结果 (2)

请输入一个整数: -64
符号取反之后的值是
64。

到目前为止我们用到的运算符都需要两个操作数，这样的运算符被称为双目运算符。在 C 语言中，还有只需要一个操作数的单目运算符，以及需要三个操作数的三目运算符。

在这里第一次出现的运算符就是单目运算符中的单目 - 运算符 (unary - operator)。可能大家都很清楚，它的功能就是对运算符进行符号取反操作。另外还有一个跟它成对的运算符——单目 + 运算符 (unary + operator)，具体请参考表 2-3。

表 2-3 单目 + 运算符和单目 - 运算符

单目 + 运算符	+a	a 的值。
单目 - 运算符	-a	对 a 进行符号取反后的值。

► 对 + 和 - 来说，存在双目和单目两个版本。单目 + 运算符实际上并没有进行什么运算，只是为了对应单目 - 运算符而准备的。

另外，单目 + 运算符、单目 - 运算符、! 运算符 (4-1 节) 和 ~ 运算符 (7-2 节) 这四个运算符统称为单目算术运算符 (unary arithmetic operator)。

赋值运算符

有一个运算符尽管到目前为止我们并没有进行特别的说明，但是之前很多程序都已经用到了。这就是表 2-4 所示的**赋值运算符**（assignment operator），即 `=`。

► `=` 一般被称为赋值运算符，但为了与第 4 章将要介绍的**复合赋值运算符**区别开，它还有一个更准确的名称——**基本赋值运算符**（simple assignment operator）。

■ 表 2-4 基本赋值运算符

基本赋值运算符 <code>a=b</code> 把 <code>b</code> 的值赋给 <code>a</code> 。
--

表达式和赋值表达式

表达式（expression）由变量和常量，以及连接它们的运算符组成。例如，对于 `vx + 32` 来说，`vx`、`32` 和 `vx + 32` 都是表达式；对于 `vc = vx + 32`（赋值表达式）来说，`vc`、`vx`、`32`、`vx + 32` 和 `vc = vx + 32` 都可以看作表达式。当然，`vc` 是赋值运算符 `=` 的第一操作数，`vx + 32` 是第二操作数。

另外，请大家记住这种使用赋值运算符的表达式被称为**赋值表达式**（assignment expression）。

表达式语句

我们在 1-1 节介绍过，C 语言规定语句必须要以分号结尾，因此前面提到的赋值表达式写成如下形式，才能成为正确的语句。

```
vc = vx + 32; /* 表达式语句 */
```

这种由表达式和分号组成的语句被称为**表达式语句**（expression statement）。

► 第 6 章会对表达式语句进行详细介绍，从下一章开始，将带领大家学习 **if** 语句和 **while** 语句等表达式语句之外的语句形式。

2-2 数据类型

求平均值

让我们来考虑一下这个问题：

读取两个整数，求出它们的平均值。

程序如代码清单 2-5 所示。

代码清单 2-5

```
/*
 * 读取两个整数，显示出它们的平均值
 */

#include <stdio.h>

int main(void)
{
    int na, nb;

    puts("请输入两个整数。");
    printf("整数 A : "); scanf("%d", &na);
    printf("整数 B : "); scanf("%d", &nb);

    printf("它们的平均值是 %d。\\n", (na + nb) / 2);

    return (0);
}
```

运行结果

请输入两个整数。

整数 A : 40

整数 B : 45

它们的平均值是 42。

将表达式 $na + nb$ 括起来的 $()$ ，是优先运算的标记。如果不加括号，表达式就变为 $na + nb / 2$ ，也就是计算 na 和 $nb / 2$ 的和。这实际上与我们平时的数学计算相同，即要遵循先乘除后加减的顺序。

► 所有运算符的优先级会在 7-4 节中进行总结。

数据类型

通过运行实例我们可以发现输出的平均值并不是 42.5 而是 42，也就是说，小数点以后的部分被舍弃了。

只处理数值的整数部分——这就是 **int 类型**（type）的特征。

int 类型和 double 类型

C 语言中以浮点数 (floating-point number) 的形式来表示实数, 浮点数有几种不同的类型, 这里我们来学习一下 **double** (双精度浮点数) 类型。让我们通过代码清单 2-6 来看看 **int** 型整数和 **double** 型浮点数之间的区别。

代码清单 2-6

```
/*
 * 整数和浮点数
 */
#include <stdio.h>

int main(void)
{
    int    nx; /* 整数 */
    double dx; /* 浮点数 */

    nx=9.99;
    dx=9.99;

    printf(" int    型变量 nx 的值: %d\n", nx);
    printf("          nx / 2 : %d\n", nx / 2);

    printf(" double 型变量 dx 的值: %f\n", dx);
    printf("          dx / 2.0 : %f\n", dx / 2.0);

    return (0);
}
```

运行结果

```
int    型变量 nx 的值: 9
          nx / 2 : 4
double 型变量 dx 的值: 9.990000
          dx/2.0 : 4.995000
```

```
/*    9    */
/*    9 / 2    */
/* 9.99    */
/* 9.99 / 2.0    */
```

我们声明一个 **int** 型变量 *nx* 和一个 **double** 型变量 *dx*, 并把 9.99 作为值赋给它们。如图 2-2 所示, 把实数值赋给 **int** 型变量时, 小数点以后的部分会被舍弃, 因此存储在 *nx* 中的值就变成了 9。

当然, 对于 *nx*/2, 也就是 9/2 来说, 由于是整数 / 整数运算, 所以结果的小数点后的部分也被舍弃了。

另外需要注意的是, 在使用 **printf** 函数输出 **double** 型值的时候, 转换说明不能使用 "%d", 而要使用 "%f"。

► 转换说明 "%f" 中的 f 就是浮点数 floating-point 的首字母。%f 默认显示小数点后 6 位数字, 变更显示位数的方法将会在后面介绍。

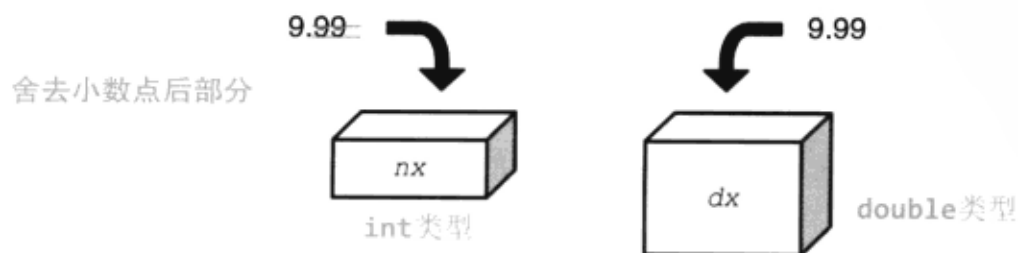


图 2-2 整数和浮点数

数据类型和对象

接下来我们进一步学习数据类型和变量。

在图 2-3 中，数据类型 **int** 和 **double** 放在虚线框中，它们对应的变量 *nx* 和变量 *dx* 放在实线框中。代表数据类型的虚线框和代表它们对应变量的实线框的大小是一样的。

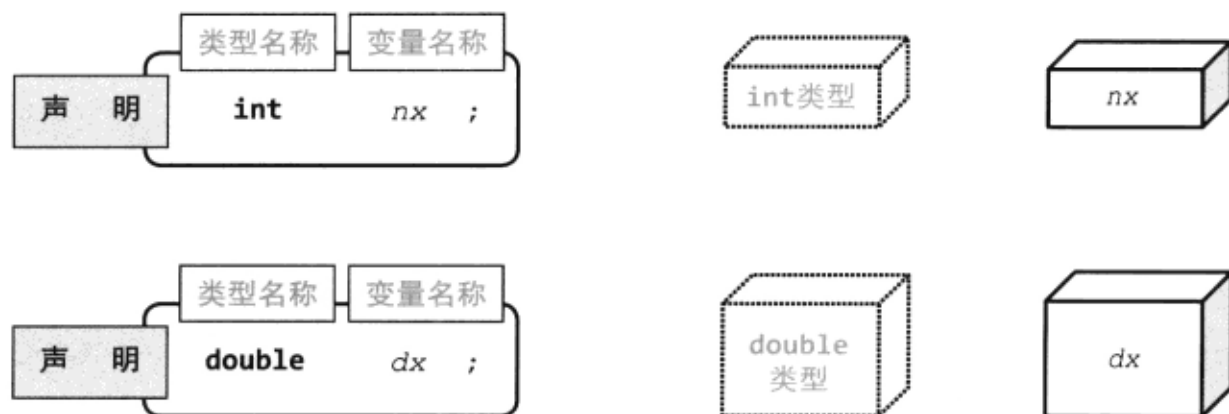


图 2-3 数据类型和对象

从前面的程序可以看出，**int** 类型只能用来存储整数，即使把实数值赋给它，也只能保留整数部分。

与之相对，浮点数中的 **double** 类型可以用来存储包含小数的实数值。

C 语言中有很多种数据类型，在第 7 章将会进行详细介绍。

不过，每种类型可存储的值都是有范围的。例如，**int** 类型的取值范围是 -32767 到 32767。

► 编译器不同，取值范围也可能更大。具体请参考 7-2 节。

这些数据类型都有一些固有的属性，继承了这些属性而创建出来的实体变量称为**对象** (object)。换句话说，我们还可以这样理解：

■ 注意 ■

数据类型实际上相当于隐藏着各种属性的一个设计蓝图（可以想象成做章鱼丸子用的模具），包含某个类型的对象（变量），就是根据这个设计蓝图创建出的实体（相当于用模具做出来的真正的章鱼烧）。

另外，“变量”这个词应用广泛，听起来比“对象”更习惯一些，加之本书对专业术语不想太过拘泥，所以本书中统一使用“变量”这个称呼。

整数常量和浮点数常量

直接在程序中指定数值的常量也有类型的区别。像 5 和 37 这样的常量，它们都是整数类型的，所以称为**整型常量**（integer constant）。像 3.14 这样包含小数的常量，称为**浮点型常量**（floating constant）。
通常整型常量都是 **int** 类型，而浮点型常量都是 **double** 类型。

► 当数值过大，或者有特殊需求的时候，也可以使用其他类型。请参考第 7 章。

double 类型的运算

编写一段程序，读取两个实数值，显示出它们的和、差、积、商。具体如代码清单 2-7 所示。

代码清单 2-7

```
/*
 * 读取两个实数值，用实数表示出它们的和、差、积、商
 */
#include <stdio.h>

int main(void)
{
    double vx, vy; /* 浮点数 */

    puts("请输入两个数。");
    printf("实数 vx :"); scanf("%lf", &vx);
    printf("实数 vy :"); scanf("%lf", &vy);

    printf("vx + vy = %f\n", vx + vy);
    printf("vx - vy = %f\n", vx - vy);
    printf("vx * vy = %f\n", vx * vy);
    printf("vx / vy = %f\n", vx / vy);

    return (0);
}
```

运行结果

请输入两个数。
实数 vx : 40.5
实数 vy : 5.2
vx + vy = 45.700000
vx - vy = 35.300000
vx * vy = 210.600000
vx / vy = 7.788462

如表 2-5 所示，**double** 类型的变量通过 **scanf** 函数赋值的时候需要使用格式字符串 **"%lf"**，请注意这一点。

表 2-5 转换说明

	int 类型	double 类型
printf	printf("%d", no)	printf("%f", no)
scanf	scanf("%d", &no)	scanf("%lf", &no)

● 练习 2-3

编写一段程序，像右面那样显示出读取的实数的值。
请输入一个实数：57.3
你输入的是57.300000。

数据类型和运算

进行整数 / 整数运算的时候，商的小数部分会被舍弃，但是浮点数之间的运算，就不会进行舍弃处理。

► 运算符 % 本身的特性决定了它只能用于整数之间的运算，而不能用于浮点数之间的运算。

如图 2-4 所示，类似 **int/int** 或者 **double/double** 这样两个类型相同的操作数之间的运算，所得结果的数据类型和运算对象的数据类型是一致的。

另外，像 **int/double** 或者 **double/int** 这种一个操作数是 **int** 类型，另一个操作数是 **double** 类型的情况，**int** 类型的操作数会进行**隐式类型转换**，自动向上转型为 **double** 类型，运算演变为 **double** 类型之间的运算。因此，运算的结果也就变成了 **double** 类型。

当然，这样的规则对于 + 或者 * 等其他运算也适用。

5	/	2	→	2		5.0	/	2.0	→	2.5
int	/	int	→	int		double	/	double	→	double

相同类型之间的运算

5.0	/	2		5	/	2.0
double	/	int		int	/	double
		↓ 向上类型转换				↓ 向上类型转换
5.0	/	2.0	→	5.0	/	2.0
double	/	double	→	double	/	double

不同类型之间的运算

图 2-4 运算和数据类型

由于 C 语言包含了很多种数据类型，详细的规则非常复杂，所以我们可以大致理解如下(详细的规则会在 7-4 节进行介绍)：

■ 注意 ■

运算对象，即操作数的类型不同时，较小的数据类型操作数会转换为较大的数据类型（范围更大），然后再进行运算。

► 所谓的“较大数据类型”，并不是说 **double** 类型实际上比 **int** 类型更大，而是说它还可以保存小数点之后部分。

让我们通过代码清单 2-8 所示的程序来验证一下这一规则。

代码清单 2-8

```

/*
 验证数据类型和运算
*/

#include <stdio.h>

int main (void)
{
    int      n1, n2, n3, n4;          /* 整数 */
    double   d1, d2, d3, d4;          /* 浮点数 */

    n1 = 5 / 2;                       /* n1 ← 2 */
    n2 = 5.0 / 2.0;                   /* n2 ← 2.5 (赋值时舍弃小数点以后的部分) */
    n3 = 5.0 / 2;                     /* n3 ← 2.5 (赋值时舍弃小数点以后的部分) */
    n4 = 5 / 2.0;                     /* n4 ← 2.5 (赋值时舍弃小数点以后的部分) */

    d1 = 5 / 2;                       /* d1 ← 2 */
    d2 = 5.0 / 2.0;                   /* d2 ← 2.5 */
    d3 = 5.0 / 2;                     /* d3 ← 2.5 */
    d4 = 5 / 2.0;                     /* d4 ← 2.5 */

    printf("n1 = %d\n", n1);
    printf("n2 = %d\n", n2);
    printf("n3 = %d\n", n3);
    printf("n4 = %d\n\n", n4);

    printf("d1 = %f\n", d1);
    printf("d2 = %f\n", d2);
    printf("d3 = %f\n", d3);
    printf("d4 = %f\n", d4);

    return (0);
}

```

运行结果

```

n1 = 2
n2 = 2
n3 = 2
n4 = 2

d1 = 2.000000
d2 = 2.500000
d3 = 2.500000
d4 = 2.500000

```

把 2 赋给 **int** 型变量 *n1*，把 2.5 分别赋给 *n2*、*n3* 和 *n4*。由于在赋值的时候会舍弃掉小数点之后的部分，因此最后这四个变量的值都是 2。

而把 2 赋给 **double** 型变量 *d1*，把 2.5 分别赋给 *d2*、*d3* 和 *d4* 的时候，它们都能把这些值完整地保存起来。

► 浮点数也存在精度的限制，并不能把数值的全部位数都表现出来。具体请参照 7-4 节。

● 练习 2-4

编写程序对整型常量、浮点型常量、**int** 型变量和 **double** 型变量进行乘除等各种运算，从而验证本节介绍的规则。

类型转换

代码清单 2-5 中计算两个整数平均值的程序，只是输出了平均值的整数部分。这次我们尝试将小数部分也一起输出。程序如代码清单 2-9 所示。

代码清单 2-9

```
/*
    读取两个整数并用浮点数显示出它们平均值
*/

#include <stdio.h>

int main(void)
{
    int na, nb;

    puts("请输入两个整数。");
    printf("整数 A :"); scanf("%d", &na);
    printf("整数 B :"); scanf("%d", &nb);

    printf("它们的平均值是 %f。\\n", (na + nb) / 2.0);

    return (0);
}
```

运行结果

请输入两个整数。
整数 A : 40
整数 B : 45
它们的平均值是 42.500000。

让我们看一下求平均值的表达式。

$(na + nb) / 2.0$ 。

首先计算的是括号内 $na + nb$ 的部分，由于 `int + int` 是整数 + 整数的运算，所以结果也是 `int` 型整数。因此表达式就变成了（整数）/2.0，也就是整数 / 浮点数，这样运算结果也就变成了浮点数。

这就计算出了平均值的浮点数结果。

但是，日常生活中计算平均值的时候，我们都会说“除以 2”，而不会说“除以 2.0”。

下面我们来看一下代码清单 2-10 中的程序：将两个整数的和转换为浮点数，然后再除以 2 计算出平均值。

代码清单 2-10

```

/*
  读取两个整数并用实数显示出它们平均值（类型转换）
*/

#include <stdio.h>

int main(void)
{
    int na, nb;

    puts("请输入两个整数。");
    printf("整数 A: "); scanf("%d", &na);
    printf("整数 B: "); scanf("%d", &nb);

    printf("它们的平均值是 %f。\\n", (double) (na + nb) / 2); /* 类型转换 */

    return (0);
}

```

运行结果

请输入两个整数。
 整数 A: 40
 整数 B: 45
 它们的平均值是 42.500000。

通常“(数据类型)表达式”形式的表达式，会把表达式的值转换为该数据类型对应的值。这样的显示转换被称为类型转换(cast)，()被称为类型转换运算符(cast operator)(表 2-6)。

■ 表 2-6 类型转换运算符

类型转换运算符 (类型名) a	把 a 的值转换为指定数据类型对应的值。
-----------------	----------------------

因此，在求平均值的时候，首先根据 **(double) (na + nb)**，把 **na + nb** 的值转换为 **double** 类型的值（例如整数 85 会转换为浮点数 85.0）。由于表达式 **(na + nb)** 的值被转换成了 **double** 类型，所以求平均值的表达式就变成了“实数 / 整数”，得到的平均值也就变成了实数。

● 练习 2-5

编写一段程序，像下面那样读取两个整数的值，计算出前者是后者的百分之几，并用实数输出结果。

请输入两个整数。
 整数A: 54
 整数B: 84
 A是B的64.285714%。

转换说明

请看代码清单 2-11 中的程序：读取三个整数，并显示出它们的合计值和平均值。

代码清单 2-11

```
/*
 * 读取三个整数，并显示出它们的合计值和平均值
 */

#include <stdio.h>

int main(void)
{
    int    na, nb, nc;
    int    sum;           /* 合计值 */
    double ave;           /* 平均值 */

    puts("请输入三个整数。");
    printf("整数 A :"); scanf("%d", &na);
    printf("整数 B :"); scanf("%d", &nb);
    printf("整数 C :"); scanf("%d", &nc);

    sum = na + nb + nc;
    ave = (double)sum / 3; /* 类型转换 */

    printf("它们的合计值是 %5d。\\n", sum); /* 99999 */
    printf("它们的平均值是 %5.1f。\\n", ave); /* 999.9 */

    return (0);
}
```

运行结果

请输入三个整数。
整数 A : 87
整数 B : 45
整数 C : 59
它们的合计值是 191。
它们的平均值是 63.7。

这个程序要注意的是：传递给 **printf** 函数的格式化字符串中的两个转换说明 **%5d** 和 **%5.1f**。它们的含义如下：

%5d ... 显示至少 5 位的十进制整数。

%5.1f ... 显示至少 5 位的浮点数。但是，小数点后只显示 1 位。

我们再详细地解释一下，转换说明的形式如下所示：

%09.9f

请对比代码清单 2-12 的运行结果来理解。

(a) 0 标志

设定了 0 标志之后，如果数值的前面有空余位，则用 0 补齐位数（如果省略了 0 标志，则会用空白补齐位数）。

(b) 最小字段宽度

也就是至少要显示出的字符位数。不设定该位数或者显示数值的实际位数超过它的时候，会根据数值显示出必要的位数。另外，如果设定了“-”，数据会左对齐显示，未设定则会右对齐显示。

代码清单 2-12

```
/*
 * 格式化整数和浮点数并显示
 */
```

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    printf("[%d]\n", 123);
```

```
    printf("[%4d]\n", 123);
```

```
    printf("[%4d]\n", 123);
```

```
    printf("[%04d]\n", 123);
```

```
    printf("[%4d]\n\n", 123);
```

```
    printf("[%d]\n", 12345);
```

```
    printf("[%3d]\n", 12345);
```

```
    printf("[%3d]\n", 12345);
```

```
    printf("[%03d]\n", 12345);
```

```
    printf("[%3d]\n\n", 12345);
```

```
    printf("[%f]\n", 123.13);
```

```
    printf("[%1f]\n", 123.13);
```

```
    printf("[%6.1f]\n\n", 123.13);
```

```
    printf("[%f]\n", 123.13);
```

```
    printf("[%1f]\n", 123.13);
```

```
    printf("[%4.1f]\n\n", 123.13);
```

```
    return (0);
```

```
}
```

运行结果

```
[123]
[0123]
[ 123]
[0123]
[123 ]

[12345]
[12345]
[12345]
[12345]
[12345]
[12345]

[123.130000]
[123.1]
[ 123.1]

[123.130000]
[123.1]
[123.1]
```

(c) 精度

指定显示的最小位数，如果不指定，则整数的时候默认为1，浮点数的时候默认为6。

(d) 转换说明符

d ... 显示十进制的 **int** 型整数。

f ... 显示十进制的 **double** 型浮点数。

► 这里介绍的只是转换说明的一部分内容，关于 **printf** 函数的详细说明请参考附录 2。

● 练习 2-6

编写一段程序，像右面那样读取身高的整数值， 请输入您的身高：175□
显示出标准体重的实数值。标准体重根据公式（身 您的标准体重是67.5公斤。
高 - 100）× 0.9 进行计算，所得结果保留一位小数。

第 3 章

分支结构程序

程序并不会总是执行同样的处理。例如，按下某个键的时候执行 A 处理，按下其他键的时候执行 B 处理……像这样，程序通过条件判断的结果选择性地执行某种处理的情况是非常多见的。

本章将会带领大家学习根据条件改变程序流程的基本方法。



3-1 if语句

程序的流程

大家的每一天都是怎样度过的呢？应该不会是日复一日地按照同样的生活模式度过吧。不管大家是否已经意识到了，其实我们都是通过某种判断来决定自己的行动的。例如，因为今天好像要下雪，所以需要给轮胎戴上防滑链。这就是一个很好的例子。

程序也是如此，几乎没有只通过预先设计好的流程就能执行的程序。本章我们将会学习通过条件来改变程序流程的方法。首先让我们来思考一下下面的问题。

如果输入的整数不能被 5 整除，就显示出相应的信息。

程序如代码清单 3-1 所示。

代码清单 3-1

```
/*
 * 输入的整数能被 5 整除吗
 */

#include <stdio.h>

int main(void)
{
    int vx;

    printf("请输入一个:");
    scanf("%d", &vx);

    if (vx % 5)
        puts("输入的整数不能整除。");

    return (0);
}
```

运行结果 (1)

请输入一个整数: 17
输入的整数不能被 5 整除。

运行结果 (2)

请输入一个整数: 15

if 语句 (1)

首先我们来看一下程序中蓝色底纹的部分。这里的 **if** 和英语中的一样，是“如果”的意思。这部分的格式如下：

if (表达式) 语句

这样的语句称为 **if 语句** (if statement)。**if** 语句会让程序执行如下处理。

判断表达式的值，如果结果不为 0 则执行相应的语句。

程序的控制流程如图 3-1 所示。

► 括号内对条件进行判断的表达式称为控制表达式 (control expression)。

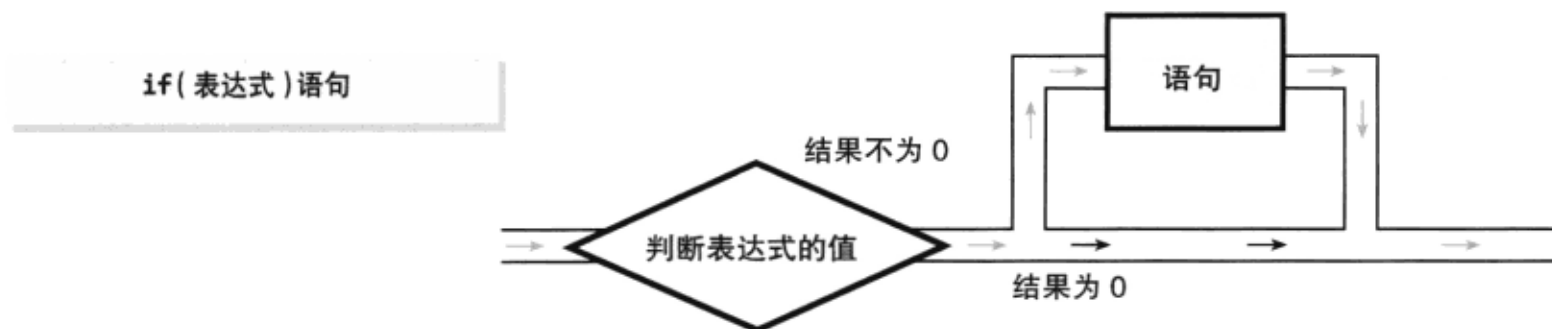


图 3-1 if 语句的流程 (1)

本例中 **if** 语句的表达式，即控制表达式是 `vx % 5`。该表达式的判断结果为 `vx` 除以 5 的余数，因此只有当这个余数不为 0，也就是 `vx` 的值不能被 5 整除的时候，才会执行后续的语句：

```
puts(" 输入的整数不能被 5 整除。");
```

而当输入的整数能被 5 整除的时候，后续语句则不会执行，屏幕上不会显示任何内容。大家也可以通过运行结果进行确认。

奇数的判定

通过判断输入的整数能否被 2 整除，就可以确认该整数是不是奇数了。程序如代码清单 3-2 所示。

代码清单 3-2

```
/*
 * 输入的整数是奇数吗
 */
#include <stdio.h>

int main(void)
{
    int no;

    printf(" 请输入一个整数: ");
    scanf("%d", &no);

    if (no % 2)
        puts(" 输入的整数是奇数。");

    return (0);
}
```

运行结果 (1)

请输入一个整数: 17
输入的整数是奇数。

运行结果 (2)

请输入一个整数: 16

if 语句 (2)

执行代码清单 3-1 中的程序时，当输入值能被 5 整除时不输出任何信息。这样很可能会让使用者不放心，所以这次我们来修改一下程序，让它在输入整数能被 5 整除的时候也显示出相应的信息。程序如代码清单 3-3 所示。

代码清单 3-3

```
/*
    输入的整数能否被 5 整除
*/

#include <stdio.h>

int main(void)
{
    int vx;

    printf("请输入一个整数:");
    scanf("%d", &vx);

    if (vx % 5)
        puts("该整数不能被5整除。");
    else
        puts("该整数能被5整除。");

    return (0);
}
```

运行结果 (1)

请输入一个整数: 17
该整数不能被 5 整除。

运行结果 (2)

请输入一个整数: 35
该整数能被 5 整除。

本程序中使用的是下面这种形式的 if 语句。

if (表达式) 语句₁ else 语句₂

这样，当表达式的值不为 0 的时候执行语句₁，当表达式的值为 0 的时候执行语句₂。这样就可以像图 3-2 所示的那样选择执行语句了。

当输入值能被 5 整除的时候，也要输出相应的信息，这样我们就能清晰地通过运行结果来判断了。

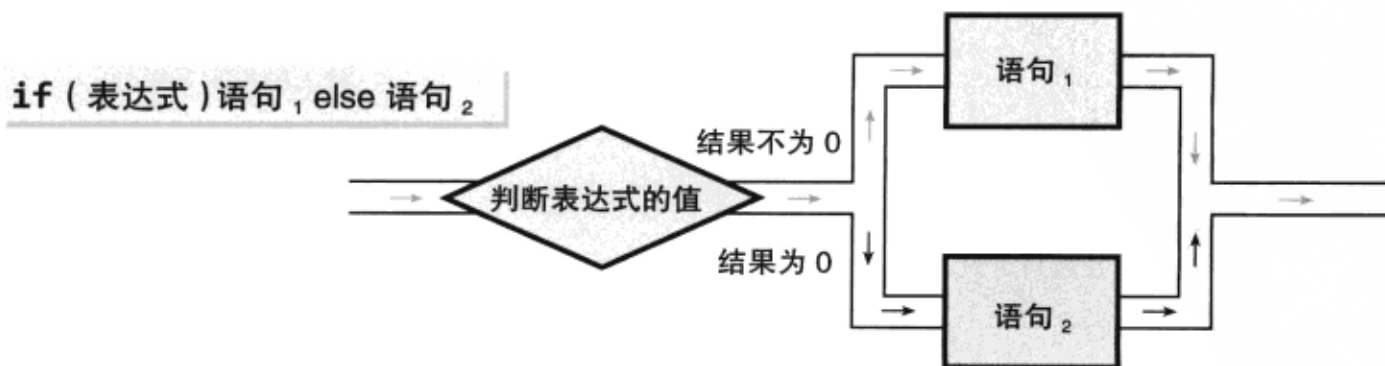


图 3-2 if 语句的流程 (2)

奇数和偶数

判断输入的整数值是奇数还是偶数并显示出来。程序如代码清单 3-4 所示。

代码清单 3-4

```
/*
    输入的整数值是奇数还是偶数
*/

#include <stdio.h>

int main(void)
{
    int no;

    printf("请输入一个整数:");
    scanf("%d", &no);

    if (no % 2)
        puts("该整数是奇数.");
    else
        puts("该整数是偶数.");

    return (0);
}
```

运行结果 (1)

请输入一个整数: 39
该整数是奇数。

运行结果 (2)

请输入一个整数: 16
该整数是偶数。

判断

在 **if** 语句的说明中，我们使用了“判断表达式”这种说法。在前一章也讲解过，变量或者常量都可以作为表达式，它们通过运算符结合起来后同样也是表达式。

C 语言的表达式，都必须能对其值进行判断（部分特殊情况除外）。例如，我们来看看下面这个表达式（假设变量 *cn* 是 **int** 类型的）。

cn + 38

当然，*cn*、38、*cn* + 38 每一个都是表达式。

当变量 *cn* 的值为 15 的时候，上述表达式的判断结果就如图 3-3 所示，分别为 15、38、53。

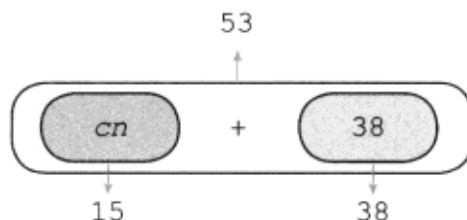


图 3-3 表达式的判断

非 0 的判断

判断输入值是否为 0 的程序如代码清单 3-5 所示。

if 语句的控制表达式，只是一个变量 *num*。由于 **if** 语句是根据控制表达式的值是否为 0 来控制程序流程的，所以对于包含变量 *num* 不为 0 和为 0 两个分支的 **if** 语句来说，这样书写更加简洁。

代码清单 3-5

```
/*
    输入的整数值是否为 0
*/

#include <stdio.h>

int main(void)
{
    int num;

    printf("请输入一个整数:");
    scanf("%d", &num);

    if (num)
        puts("该整数不是0。");
    else
        puts("该整数是0。");

    return (0);
}
```

运行结果 (1)

请输入一个整数: 15
该整数不是 0。

运行结果 (2)

请输入一个整数: 0
该整数是 0。

语法结构

截至目前，我们用到了两种 **if** 语句，其结构如图 3-4 所示（结构图的详细说明请参考专题 3-1）。

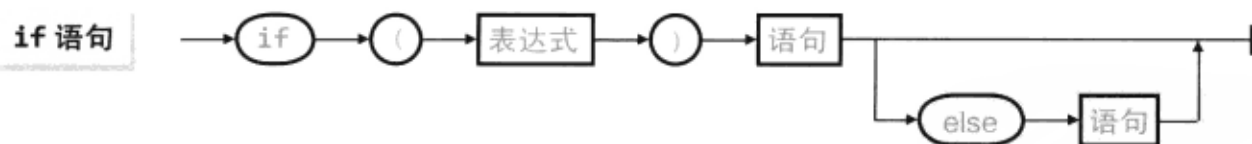


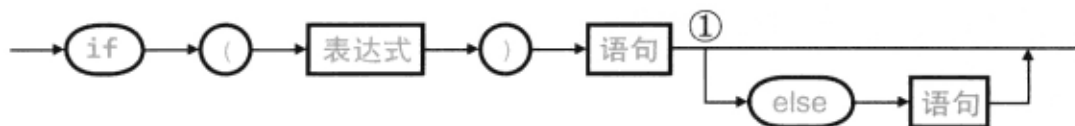
图 3-4 if 语句的结构图

如果不满足上述结构就会出现错误。例如下面两个语句在编译的时候会发生错误，当然也就无法执行。

```
if va % vb puts("va 不能被 vb 整除。"); /* 表达式缺少括号 */
if (cx / dx) else d = 3; /* 缺少最开始的语句 */
```

专题 3-1 语法结构图

本书中使用的结构图都是通过箭头把各个元素连接起来的。



关于元素

结构图中的元素，既有用圆形表示的，也有用方形表示的。

圆形：像 **if** 这样的关键字或者 “(” 这样的分隔符，都必须按照这种形式书写，不能写成 “如果” 或 “[”。这样的内容使用圆形表示。

方形：这里所说的关键字和分隔符以外的**表达式**或者**语句**，在程序中并不直接写作“表达式”，而是像 `a + b` 这样写成具体的表达式。像这种不能直接书写的语法概念的内容使用方形表示。

※ 关键字和分隔符的概念会在第 4 章进行说明。

结构图的阅读方法

阅读结构图的时候，要沿着箭头的走向理解，从左边开始，到右边结束。遇到分支点，可以选择任意分支继续理解。

对于上图中的分支点①来说，由于任意分支都可以继续下去，因此 **if** 语句的结构图从左到右的路径有以下两种：

if (表达式) 语句

if (表达式) 语句 else 语句

这就是 **if** 语句的格式，或者说结构。

例如，代码清单 3-1 中的 **if** 语句如下所示。

```
if ( 表达式 )          语句
if ( vx % 5 ) puts ( "该整数不能被 5 整除。");
```

代码清单 3-3 中的 **if** 语句如下所示。

```
if ( 表达式 )          语句          else          语句
if ( vx % 5 ) puts ( "该整数不能被 5 整除。"); else puts ( "该整数能被 5 整除。");
```

它们都符合 **if** 语句的语法结构。

● 练习 3-1

编写一段程序，像右面这样输入两个整数值，如果后者是前者的约数，则显示“B 是 A 的约数”。如果不是，则显示“B 不是 A 的约数”。

请输入两个整数。

整数A: 12

整数B: 6

B是A的约数。

相等运算符

接下来让我们考虑一下下面的问题。

输入两个整数值，判断它们是否相等。

程序如代码清单 3-6 所示。

代码清单 3-6

```
/*
 * 输入的两个整数相等吗
 */
#include <stdio.h>

int main(void)
{
    int x1, x2;

    puts("请输入两个整数。");
    printf("整数 1:"); scanf("%d", &x1);
    printf("整数 2:"); scanf("%d", &x2);

    if (x1 == x2)
        puts("它们相等。");
    else
        puts("它们不相等。");

    return (0);
}
```

运行结果 (1)

请输入两个整数。
整数 1: -5
整数 2: 5
它们相等。

运行结果 (2)

请输入两个整数。
整数 1: 40
整数 2: 45
它们不相等。

让我们来看一下 **if** 语句的控制表达式。本书中第一次出现的 **== 运算符**，会对左右两侧的操作数进行比较，如果它们相等则结果为 1，如果不相等则结果为 0（结果是 **int** 型整数）。在上述程序中，如果 **x1** 和 **x2** 相等，表达式 **x1 == x2** 的值为 1，否则为 0。因此，当输入的两个整数值相等时，执行如下语句：

```
puts(" 它们相等。");
```

当输入的值不相等时，执行如下语句：

```
puts(" 它们不相等。");
```

就像表 3-1 中总结的那样，与 **==** 运算符相反，用来判断两个操作数是否不相等的是 **!= 运算符**。这两种运算符统称为**相等运算符**（equality operator）。

表 3-1 相等运算符

== 运算符	a==b	如果 <i>a</i> 和 <i>b</i> 的值相等则为 1，不等则为 0（结果的类型是 int ）。
!= 运算符	a!=b	如果 <i>a</i> 和 <i>b</i> 的值不相等则为 1，相等则为 0（结果的类型是 int ）。

使用 `!=` 运算符修改后的程序如代码清单 3-7 所示。虽然程序的内容有所不同，但是结果却完全一样。

代码清单 3-7

```
/*
 * 输入的两个整数相等吗（第二版）
 */

#include <stdio.h>

int main(void)
{
    int x1, x2;

    puts("请输入两个整数。");
    printf("整数 1:"); scanf("%d", &x1);
    printf("整数 2:"); scanf("%d", &x2);

    if (x1 != x2)
        puts("它们不相等。");
    else
        puts("它们相等。");

    return (0);
}
```

运行结果 (1)

请输入两个整数。
整数 1: 5
整数 2: 5
它们相等。

运行结果 (2)

请输入两个整数。
整数 1: 40
整数 2: 45
它们不相等。

比较余数

判定输入整数的个位数是否为 5 并显示相应信息的程序如代码清单 3-8 所示。

代码清单 3-8

```
/*
 * 个位数是 5 吗
 */

#include <stdio.h>

int main(void)
{
    int vx;

    printf("请输入一个整数:");
    scanf("%d", &vx);

    if (vx % 10 == 5)
        puts("该整数的个位数是5。");

    return (0);
}
```

运行结果 (1)

请输入一个整数: 15
该整数的个位数是 5。

运行结果 (2)

请输入一个整数: 59

► 表达式 `vx % 10 == 5` 如果写成 `(vx % 10) == 5` 会更容易理解。

关系运算符

到目前为止，我们已经见过了包含两个分支的程序流程，现在来看看三个分支的情况。请大家考虑一下下面的问题。

输入一个整数，判断该整数的符号。

程序如代码清单 3-9 所示。

代码清单 3-9

```

/*
判断输入整数的符号
*/

#include <stdio.h>

int main(void)
{
    int no;

    printf("请输入一个整数:");
    scanf("%d", &no);

    if (no == 0)
        puts("该整数为0。");
    else if(no > 0)
        puts("该整数为正数。");
    else
        puts("该整数为负数。");

    return (0);
}
    
```

运行结果 (1)

请输入一个整数: 0
该整数为 0。

运行结果 (2)

请输入一个整数: 39
该整数为正数。

运行结果 (3)

请输入一个整数: -4
该整数为负数。

在本程序中我们第一次接触到了 **> 运算符**，该运算符对左右两侧操作数的值进行比较，如果第一操作数大于第二操作数，则结果为 1，反之结果为 0（结果为 **int** 型整数）。也就是说，如果 *no* 大于 0，表达式 *no > 0* 的结果为 1，否则为 0。

比较两个操作数大小关系的运算符称为**关系运算符**（relational operator），该类运算符如表 3-2 所示共有四种。

表 3-2 关系运算符

< 运算符	<i>a</i> < <i>b</i>	<i>a</i> 小于 <i>b</i> 时结果为 1，反之为 0（结果的类型为 int ）。
> 运算符	<i>a</i> > <i>b</i>	<i>a</i> 大于 <i>b</i> 时结果为 1，反之为 0（结果的类型为 int ）。
<= 运算符	<i>a</i> <= <i>b</i>	<i>a</i> 小于等于 <i>b</i> 时结果为 1，反之为 0（结果的类型为 int ）。
>= 运算符	<i>a</i> >= <i>b</i>	<i>a</i> 大于等于 <i>b</i> 时结果为 1，反之为 0（结果的类型为 int ）。

► 请大家注意，将 <= 运算符和 >= 运算符中的等号放在左侧(=< 和 =>)，或者在 < 与 = 之间有空格都是不对的。

嵌套的 if 语句

通过之前的介绍，相信大家已经明白了程序的功能。但是，为了更准确地加以理解，还是有必要对语法进行说明。

如前所述，**if** 语句有以下两种形式

if (表达式) 语句

if (表达式) 语句 else 语句

虽然本程序中用到了 `...else if...`，但这并不是标准的语法结构。**if** 语句，顾名思义是一种语句，因此 **else** 控制的语句也可以是 **if** 语句。

如图 3-5 那样书写应该就可以理解了吧。**if** 语句中嵌入 **if** 语句，就形成了嵌套结构。

```
if (no == 0)
    语句 ..... puts ("该整数为 0。");
else
    语句 ..... if (no > 0)
                  语句 ..... puts ("该整数为正数。");
                  else
                      语句 ..... puts ("该整数为负数。");
```

图 3-5 嵌套的 if 语句

● 练习 3-2

编写一段程序，确认相等运算符和关系运算符的运算结果是 1 和 0。

● 练习 3-3

编写一段程序，像右面这样输入一个整数值，显示出它的绝对值。

请输入一个整数: -8
绝对值是8。

● 练习 3-4

编写一段程序，像右面这样输入两个整数，如果两数值相等，则显示“A 和 B 相等。”。如果 A 大于 B，则显示“A 大于 B。”。如果 A 小于 B，则显示“A 小于 B。”。

请输入两个整数。
整数A: 12
整数B: 6
A大于B。

计算较大值

输入两个整数，显示出其中较大的值。具体程序如代码清单 3-10 所示。

代码清单 3-10

```
/*
 * 显示出输入的两个整数中较大的数
 */

#include <stdio.h>

int main(void)
{
    int n1, n2;

    puts("请输入两个整数。");
    printf("整数 1 : "); scanf("%d", &n1);
    printf("整数 2 : "); scanf("%d", &n2);

    if (n1 > n2)
        printf("较大的数是%d。\\n", n1);
    else
        printf("较大的数是%d。\\n", n2);

    return (0);
}
```

运行结果

请输入两个整数。
整数 1 : 83
整数 2 : 45
较大的数是 83。

本程序中，同一 **printf** 函数被调用了两次。下面我们来改写一下，首先把两数中较大的值存入变量，然后再进行显示。程序如代码清单 3-11 所示。

代码清单 3-11

```
/*
 * 显示出输入的两个整数中较大的数
 */

#include <stdio.h>

int main(void)
{
    int n1, n2, max;

    puts("请输入两个整数。");
    printf("整数 1 : "); scanf("%d", &n1);
    printf("整数 2 : "); scanf("%d", &n2);

    if (n1 > n2) max = n1; else max = n2;

    printf("较大的数是%d。\\n", max);

    return (0);
}
```

运行结果

请输入两个整数。
整数 1 : 83
整数 2 : 45
较大的数是 83。

为了让求出的较大的数在后续程序中可以使用，可以像上述程序那样把它保存在变量中。

计算三个数的最大值

这次我们输入三个整数，显示出其中的最大值。程序如代码清单 3-12 所示。

代码清单 3-12

```
/*
 * 计算输入的三个整数中的最大值并显示
 */
#include <stdio.h>

int main(void)
{
    int n1, n2, n3, max;

    puts("请输入三个整数。");
    printf("整数 1:"); scanf("%d", &n1);
    printf("整数 2:"); scanf("%d", &n2);
    printf("整数 3:"); scanf("%d", &n3);

    max = n1;
    if (n2 > max) max = n2;
    if (n3 > max) max = n3;

    printf("最大值是 %d。\\n", max);

    return (0);
}
```

运行结果

请输入三个整数。
 整数 1: 83
 整数 2: 45
 整数 2: 62
 最大值是 83。

请大家注意一下程序中蓝色底纹部分。

- (1) 把 *n1* 的值保存到变量 *max* 中。
- (2) 如果 *n2* 的值大于 *max*，则将 *n2* 的值赋给变量 *max*。

※ 如果 *n2* 小于 *max*，则不对 *max* 赋值

- (3) 对 *n3* 进行与 (2) 相同的处理

完成以上操作之后，变量 *max* 中的值就是 *n1*、*n2*、*n3* 中的最大值了。

● 练习 3-5

编写一段程序，计算出输入的三个整数中的最小值并显示。

● 练习 3-6

编写一段程序，计算出输入四个整数中的最大值并显示。

条件运算符

表 3-3 介绍的是与 **if** 语句非常相似的条件运算符 (conditional operator)。该运算符是需要三个操作数的三目运算符。

- ▶ 只有条件运算符属于三目运算符，其他的运算符都是单目或者双目运算符。

■ 表 3-3 条件运算符

条件运算符	$a ? b : c$	如果 a 不为 0，则结果是 b 的值，否则结果为 c 的值。
-------	-------------	---------------------------------------

使用该运算符，对代码清单 3-11 中的程序（计算输入的两个整数中较大的数）进行修改的结果如代码清单 3-13 所示。

代码清单 3-13

```
/*
 * 计算输入的两个整数中较大的数并显示（条件运算符）
 */

#include <stdio.h>

int main(void)
{
    int n1, n2, max;

    puts("请输入两个整数。");
    printf("整数 1:");      scanf("%d", &n1);
    printf("整数 2:");      scanf("%d", &n2);

    max = (n1 > n2) ? n1 : n2;    /* 将较大的值赋给 max */

    printf("较大的数是 %d.\n", max);

    return (0);
}
```

运行结果

请输入两个整数。
整数 1: 83
整数 2: 45
较大的数是 83。

接下来我们对条件运算符中使用的条件表达式 (conditional expression) 进行一些具体的说明。

表达式₁ ? 表达式₂ : 表达式₃

对上述条件表达式进行判断的结果如下所示：

首先，判断表达式₁的值。

- 如果不为 0，则条件表达式整体的值为表达式₂的值。
- 如果为 0，则条件表达式整体的值为表达式₃的值。

- ▶ 如果表达式₁的判断结果不为 0，则无需执行表达式₃。如果判断结果为 0，则无需执行表达式₂。

本程序中条件表达式 $(n1 > n2) ? n1 : n2$ 的判断流程如下所示:

$n1$ 大于 $n2$ 的时候

表达式₁ $(n1 > n2)$ 的判断结果为 1, 表达式整体的判断结果为表达式₂ 中 $n1$ 的值。

$n1$ 小于等于 $n2$ 的时候

表达式₁ $(n1 > n2)$ 的判定结果为 0, 表达式整体的判断结果为表达式₃ 中 $n2$ 的值。

这样, 就可以通过该条件表达式, 把 $n1$ 和 $n2$ 中较大的值保存到变量 `max` 中。

差值计算

使用条件运算符计算输入的两个整数差值的程序如代码清单 3-14 所示。

代码清单 3-14

```
/*
    计算输入的两个整数的差并显示 (条件运算符)
*/

#include <stdio.h>

int main(void)
{
    int n1, n2;

    puts("请输入两个整数。");
    printf("整数 1: ");      scanf("%d", &n1);
    printf("整数 2: ");      scanf("%d", &n2);

    printf("它们的差是 %d.\n", (n1 > n2) ? n1 - n2 : n2 - n1);

    return (0);
}
```

运行结果

请输入两个整数。
整数 1: 83
整数 2: 45
它们的差是 38。

● 练习 3-7

使用 `if` 语句替换代码清单 3-14 程序中的条件运算符, 实现同样的功能。

● 练习 3-8

使用条件运算符替换练习 3-5 程序中的 `if` 语句, 实现同样的功能。

复合语句（程序块）

计算输入的两个整数中较大值和较小值的程序如代码清单 3-15 所示。

代码清单 3-15

```
/*
 * 计算输入的两个整数中较大数和较小数并显示
 */

#include <stdio.h>

int main(void)
{
    int n1, n2, max, min;

    puts("请输入两个整数。");
    printf("整数 1 : "); scanf("%d", &n1);
    printf("整数 2 : "); scanf("%d", &n2);

    if (n1 > n2) {
        max = n1;
        min = n2;
    } else {
        max = n2;
        min = n1;
    }

    printf("较大的数是 %d。\\n", max);
    printf("较小的数是 %d。\\n", min);

    return (0);
}
```

运行结果

请输入两个整数。
 整数 1 : 83
 整数 2 : 45
 较大的数是 83。
 较小的数是 45。

请大家关注一下程序中蓝色底纹部分。对于该 **if** 语句，当 $n1 > n2$ 的时候执行下面语句：

```
{ max = n1; min = n2; }
```

否则，执行下面语句：

```
{ max = n2; min = n1; }
```

上文中大括号内并排书写的语句称为**复合语句**（compound statement）或者**程序块**（block）。复合语句的结构如图 3-6 所示，其中不仅可以包含语句，也可以包含声明（但是一定要把声明放在最开始的位置^①）。

复合语句

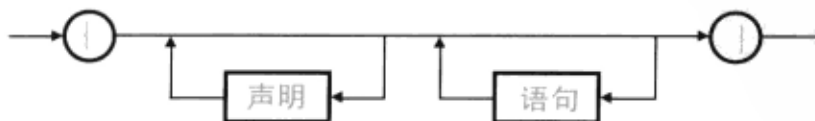


图 3-6 复合语句的结构图

① C99标准允许在任何地方定义变量。

► 简单来说就是 { 0 个以上的声明 0 个以上的语句 } 这样的结构。

例如, 以下这些全都属于复合语句。

<code>{ }</code>	<code>{ }</code>
<code>{ printf("ABC\n"); }</code>	<code>{ 语句 }</code>
<code>{ int x; x = 5; }</code>	<code>{ 声明 语句 }</code>
<code>{ int x; x = 5; printf("%d", x); }</code>	<code>{ 声明 语句 语句 }</code>
<code>{ int x; int y; x = 5; y = 3; printf("%d", x); }</code>	<code>{ 声明 声明 语句 语句 语句 }</code>

由大括号中的声明和语句组成的复合语句, 在结构上可以看作单一的语句。因此, 本程序中的 **if** 语句也可以解释成下面这样。

<code>if (条件表达式)</code>	<code>语句</code>	<code>else</code>	<code>语句</code>
<code>if (n1 > n2)</code>	<code>{ max = n1; min = n2; }</code>	<code>else</code>	<code>{ max = n2; min = n1; }</code>

在这里, 让我们回想一下 **if** 语句的语法结构。**if** 语句的形式为下列形式中之一。

if (表达式) 语句
if (表达式) 语句 **else** 语句

也就是说, **if** 语句是控制流程的语句, 只能有一种选择结果 (**else** 也只有一种选择)。所以说本程序中的 **if** 语句完全符合这样的语法结构。

如果程序写成下面这样会出现什么情况呢?

<code>if 语句</code>	<code>语句</code>	<code>else</code>	<code>语句</code>	<code>语句</code>
<code>if (n1 > n2)</code>	<code>max = n1; min = n2;</code>	<code>else</code>	<code>max = n2; min = n1;</code>	

由于编译器会把蓝色底纹部分看成一个 **if** 语句, 所以编译的时候会出现错误 (没有与 **else** 相对应的 **if**)。

■ 注意 ■

在需要单一语句的位置, 如果一定要使用多个语句, 可以把它们组合成复合语句 (程序块) 来实现。

图 3-7 中蓝色底纹部分是 C 语言的固定结构, 请大家牢记它的拼写方法 (在第 1 章中也提出过这样的建议)。

大家可以看到, 图中蓝色字体的部分就是一个复合语句。

虽然直到现在我们才第一次对复合语句进行说明, 但其实大家从最初的程序开始就已经一直在使用复合语句了。

```
#include <stdio.h>

int main(void)
{
    printf("%d", 15 + 37);

    return (0);
}
```

图 3-7 程序和固定结构

判断季节

这次让我们来思考下面的问题：

显示出输入月份所处的季节。

也就是说，根据输入的整数月份，显示成下面这样（其中 x 代表月份的数值）。

3,	4,	5	x 月是春季
6,	7,	8	x 月是夏季
9,	10,	11	x 月是秋季
12,	1,	2	x 月是冬季

如果可以像代码清单 3-9 中的程序那样恰当地使用 **if... else if...** 语句，就可以实现这样的功能。程序如代码清单 3-16 所示。

逻辑运算符

这里使用的 **&&** 称为**逻辑与运算符**（logical AND operator）。对使用该运算符的表达式 $a \&\& b$ 进行判断，如果 a 和 b 的值都不为 0，则结果为 1，否则结果为 0（结果的类型为 **int**）。大家可以把它理解为 a 并且 b 。**if** 语句最开始的判断表达式为：

```
month >= 3 && month <= 5
```

当 $month$ 的值大于等于 3 并且小于等于 5 的时候，判断结果为 1，继续执行下述语句。

```
puts ("是春天。");
```

对夏天和秋天的判断也是如此。

下面我们来看一下对冬天的判断。

```
month == 1 || month == 2 || month == 12
```

这里用到的 **||** 称为**逻辑或运算符**（logical OR operator），对使用该运算符的表达式 $a || b$ 进行判断，如果 a 和 b 都为 0，则表达式的结果为 0，否则结果为 1（结果的类型为 **int**）。大家可以把它理解为 a 或者 b 。

► 我们平时说到“我或者他会去”的时候，表示“我”和“他”中只有一个人会去。但是对于 **||** 运算符来说，表达的却是至少有一个即可的意思，请大家特别注意。

代码清单 3-16

```

/*
    显示输入月份所处的季节
*/

#include <stdio.h>

int main(void)
{
    int month;          /* 月份 */

    printf("请输入月份:");
    scanf("%d", &month);

    if (month >= 3 && month <= 5)
        puts("是春天。");
    else if (month >= 6 && month <= 8)
        puts("是夏天。");
    else if (month >= 9 && month <= 11)
        puts("是秋天。");
    else if (month == 1 || month == 2 || month == 12)
        puts("是冬天。");
    else
        puts("输入月份不存在!!\a");

    return (0);
}
    
```

运行结果 (1)

请输入月份: 5
是春天。

运行结果 (2)

请输入月份: 8
是夏天。

► 大家是否还记得第 1 章中提到的用来发出警报的转义字符 \a 呢?

表 3-4 逻辑运算符

逻辑与运算符 $a \&\& b$	如果 a 和 b 都不为 0, 则表达式的结果为 1, 否则结果为 0 (结果的类型为 <code>int</code>)。
逻辑或运算符 $a b$	如果 a 和 b 都为 0, 则表达式的结果为 0, 否则结果为 1 (结果的类型为 <code>int</code>)。

● 练习 3-9

编写一段程序, 像右面这样输入三个整数, 如果三个数都相等, 则显示“三个值都相等”。如果其中任意两个值相等, 则显示“有两个值相等”。如果上述两种情况都不满足, 则显示“三个值各不相同”。

请输入三个整数。
整数 A: 12
整数 B: 6
整数 C: 12
有两个值相等。

● 练习 3-10

编写一段程序, 像右面这样输入两个整数, 如果它们的差值小于等于 10, 则显示“它们的差小于等于 10”。否则, 显示“它们的差大于等于 11”。

请使用逻辑或运算符。

请输入两个整数。
整数 A: 12
整数 B: 6
它们的差小于等于 10。

3-2 switch语句

程序的流程

显示输入整数除以 3 所得余数的程序如代码清单 3-17 所示。

代码清单 3-17

```
/*
 * 显示出输入整数除以 3 的余数
 */

#include <stdio.h>

int main(void)
{
    int num;

    printf("请输入一个整数:");
    scanf("%d", &num);

    if (num % 3 == 0)
        puts("该数能被 3 整除。");
    else if (num % 3 == 1)
        puts("该数除以 3 的余数是 1。");
    else
        puts("该数除以 3 的余数是 2。");

    return (0);
}
```

运行结果 (1)

请输入一个整数: 6
该数能被 3 整除。

运行结果 (2)

请输入一个整数: 40
该数除以 3 的余数是 1。

本程序中使用了两次用来计算 num 除以 3 的余数的表达式 $num \% 3$ ，稍显冗长。

通过某一单一表达式的值，将程序分为多个分支的时候，可以使用 **switch** 语句，这样能让程序更简洁。

switch 语句的语法结构如图 3-8 所示，括号内的控制表达式必须是整数类型。

switch 语句



图 3-8 switch 语句的结构图

使用 **switch** 语句修改后的程序如代码清单 3-18 所示。**switch** 语句首先对表达式的值进行判断，然后程序会转向 **case** 后书写的值与判断结果相等的部分。

代码清单 3-18

```
/*
    显示出输入整数除以 3 的余数 (switch 语句)
*/

#include <stdio.h>

int main(void)
{
    int num;

    printf(" 请输入一个整数:");
    scanf("%d", &num);

    switch (num % 3) {
        case 0 : puts(" 该数能被 3 整除。");      break;
        case 1 : puts(" 该数除以 3 的余数是 1。"); break;
        case 2 : puts(" 该数除以 3 的余数是 2。"); break;
    }

    return (0);
}
```

运行结果 (1)

请输入一个整数: 6
该数能被 3 整除。

运行结果 (2)

请输入一个整数: 40
该数除以 3 的余数是 1。

例如, 如果 $num \% 3$ 的值为 1, 则程序会转向如下部分:

case 1:

然后依次执行后续的语句。

但是, 当程序执行到被称为 **break** 语句 (break statement) 的 **break;** 时, 会跳出 **switch** 语句。**break** 语句的结构图如图 3-9 所示。

break 语句



图 3-9 break 语句的结构图

于是, 本程序的流程就变为图 3-10 所示的那样。

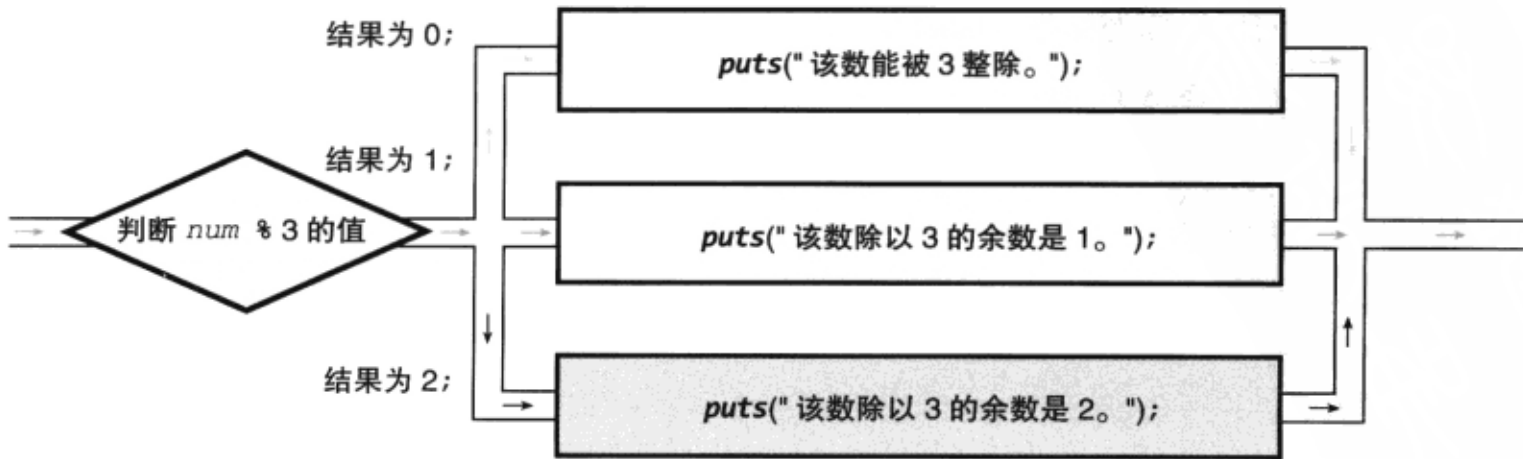


图 3-10 代码清单 3-18 中程序的流程

我们再来通过代码清单 3-19 中的程序仔细确认一下 **switch** 语句的动作。

代码清单 3-19

```

/*
    确认 switch 语句动作的程序
*/

#include <stdio.h>

int main(void)
{
    int sw;

    printf(" 请输入一个整数:");
    scanf("%d", &sw);

    switch (sw) {
        case 1 : puts("A");      puts("B");      break;
        case 2 : puts("C");
        case 5 : puts("D");      break;
        case 6 :
        case 7 : puts("E");      break;
        default : puts("F");     break;
    }

    return (0);
}
    
```

运行结果 (1)

请输入一个整数: 1
A
B

运行结果 (2)

请输入一个整数: 2
C
D

运行结果 (3)

请输入一个整数: 3
F

运行结果 (4)

请输入一个整数: 5
D

运行结果 (5)

请输入一个整数: 6
E

这次的 **switch** 语句中出现了之前程序中没有的
default :

标记。当控制表达式的判断结果与任何一个 **case** 后的值都不一致的时候，程序就会跳转到该标识继续执行。

另外，像 **case ** :** 或者 **default :** 这样用来表示程序跳转的标识称为**标签 (label)**。

本程序的流程如图 3-11 所示。

► 如果改变本程序 **switch** 语句中标签的顺序，程序的执行结果也会发生改变，所以在使用 **switch** 语句的时候，一定要正确书写标签的顺序。另外，相同的标签如果出现两次，编译的时候就会发生错误。

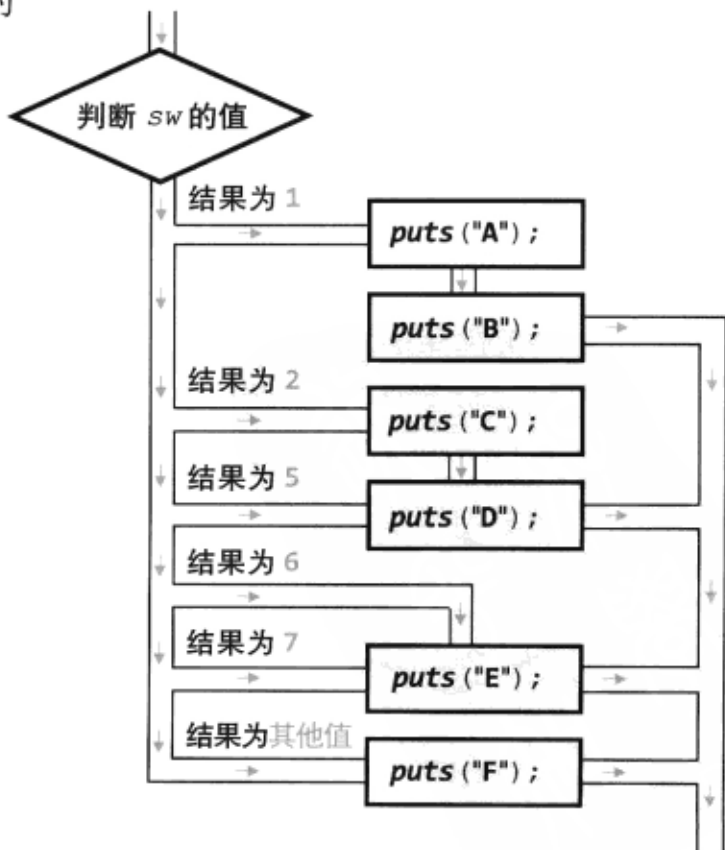


图 3-11 代码清单 3-19 中程序的流程

switch 语句和 if 语句

请大家看一下左下方所示的 **if** 语句。能实现同样功能的 **switch** 语句如右下方所示。

```
if ( va == 1)
    vc = 10;
else if (va == 2)
    vc = 20;
else if (va == 3)
    vc = 50;
else if (vb == 4)
    vc = 80;
```

```
/* 对左侧 if 语句进行修改的结果 */
switch (va) {
    case 1 : vc = 10; break;
    case 2 : vc = 20; break;
    case 3 : vc = 50; break;
    default : if (vb == 4) vc = 80;
}
```

前三个 **if** 语句，对 *va* 的值进行判断，最后一个 **if** 语句对 *vb* 的值是否为 4 进行判断。也就是说，当 *va* 不是 1、2、3 中任何一个，并且 *vb* 为 4 的时候，会把 80 赋给 *vc*。

对于连续的 **if** 语句来说，实现分支操作的比较对象并不仅仅限于单一的表达式。也就是说，可以通过对复杂条件的判断来实现一连串的分支处理。

肯定会有人把 **if** 语句的最后一个判断看成 **if**(*va*==4) 或者在书写的时候写成 **if**(*va*==4)。从这个方面来看，**switch** 语句的格式更加清晰，阅读程序的人就会很少遇到上述问题。

■ 注意 ■

通过单一表达式来控制程序流程分支的时候，通常使用 **switch** 语句的效果要比使用 **if** 语句更好。

选择语句

本章学习的 **if** 语句和 **switch** 语句，都是用来实现程序流程的选择性分支的，因此统称为选择语句 (selection statement)。

● 练习 3-11

对代码清单 3-4 中的程序进行修改，不使用 **if** 语句，而是改用 **switch** 语句来实现。

● 练习 3-12

对代码清单 3-16 中的程序进行修改，不使用 **if** 语句，而是改用 **switch** 语句来实现。



第 4 章

程序的循环控制

人生就是日复一日地不断重复，既有相同的事情，也有相似的事情，却无论如何也无法回到最初。要想在生活的每一刻都能有新的发现，恐怕只是一个美好的愿望。

本章将会为大家介绍程序中的重复流程——循环。



4-1 do语句

do 语句

首先我们对上一章介绍的代码清单 3-4 中的程序（显示出输入的整数是奇数还是偶数）进行如下修改。

输入一个整数，显示出它是奇数还是偶数。然后询问是否重复同样的操作，并按要求进行处理。

修改之后，无需重新启动，我们就可以按照自己的意愿循环执行该程序了（见代码清单 4-1）。

代码清单 4-1

```

/*
    输入的整数是奇数还是偶数呢（按照自己的意愿进行循环操作）
*/

#include <stdio.h>

int main(void)
{
    int cont;    /* 要继续吗 */

    do {
        int no;

        printf("请输入一个整数:");
        scanf("%d", &no);

        if (no % 2)
            puts("这个整数是奇数。");
        else
            puts("这个整数是偶数。");

        printf("要重复一次吗? 【Yes...0 / No...9】:");
        scanf("%d", &cont);
    } while (cont == 0);

    return (0);
}

```

运行结果

请输入一个整数: 17

这个整数是奇数。

要重复一次吗? 【Yes...0/No...9】: 0

请输入一个整数: 8

这个整数是偶数。

要重复一次吗? 【Yes...0/No...9】: 9

程序中带底色的部分可以按照自己的意愿任意循环执行，这就是 **do 语句**（do statement），其结构如图 4-1 所示。



图 4-1 do 语句的结构图

根据 **do** 语句的处理流程，只要控制表达式的结果不是 0，循环体（loop body）中的语句就会循环执行。程序的处理流程如图 4-2 所示。

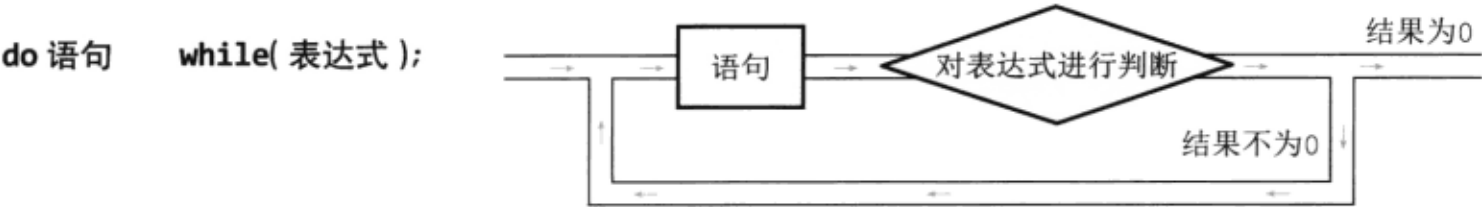


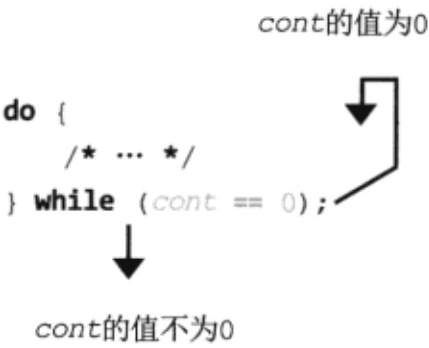
图 4-2 do 语句的处理流程

如果读取到的变量 *cont* 的值为 0，就会循环执行 **do** 和 **while** 之间循环体中的复合语句。也就是说，当 *cont* 的值是 0 的时候，程序就会返回复合语句的开头重新执行。

如果读取到的 *cont* 值不为 0，**do** 语句就结束了。

这样就能够按自己的意愿循环执行该程序了。

► 因此，读取到的 *cont* 的值即便不是 9，循环也会结束。



复合语句（程序块）中的声明

上例中的变量 *no* 是在 **do** 语句中的复合语句部分进行声明的。需要注意的是，仅在复合语句中使用的变量通常要在复合语句中进行声明。

■ 注意 ■

仅在复合语句中使用的变量要在该复合语句中进行声明。

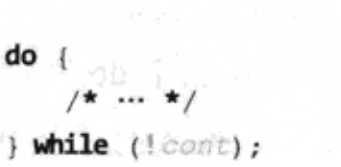
► 复合语句是由 { 0 个以上的声明 0 个以上的语句 } 组成的（请参考第 3 章）。

逻辑非运算符

我们对 **do** 语句的判断式进行一下修改（如右图所示），它还是能进行同样的处理。

“!” 运算符称为**逻辑非运算符**（logical negation operator），用于判断操作数是否为 0。

正如表 4-1 所述，*!cont* 和 *cont == 0* 具有相同的含义。



■ 表 4-1 逻辑非运算符

逻辑非运算符	!a	当a的值是0的时候值为1，当a的值不是0的时候值为0（它的结果是int类型）。
--------	----	---

逆向显示整数值

我们来考虑一下下面这个问题。

输入一个非负整数，并进行逆向显示。

也就是说，当输入 1963 的时候，显示出的结果是 3691。当输入负整数的时候，显示再次输入的提示信息。程序如代码清单 4-2 所示。

代码清单 4-2

```
/*
 * 逆向显示输入的非负整数
 */
#include <stdio.h>

int main(void)
{
    int num;

    do {
        printf("请输入一个非负整数:");
        scanf("%d", &num);
        if (num < 0);
            puts("请不要输入负整数。");
    } while (num < 0);

    printf("该整数逆向显示的结果是 ");
    do {
        printf("%d", num % 10);
        num = num / 10;
    } while (num > 0);
    puts("");

    return (0);
}
```

运行结果

请输入一个非负整数: -17

请不要输入负整数。

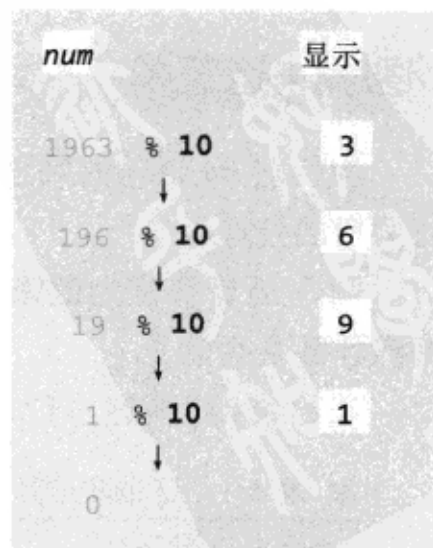
请输入一个非负整数: 1963

该整数逆向显示的结果是 3691。

本例中使用了 2 个 **do** 语句。第一个 **do** 语句只有在 *num* 的值为负的时候才循环执行，因此，当循环结束的时候 *num* 的值肯定是非负的，也就是大于等于 0。

第二个 **do** 语句用来实现输入整数的逆向显示。如右图所示，首先将输入的整数 1963 保存到变量 *num* 中，之后的处理流程如下：

先求出 *num* % 10 的余数（请参考第 2 章），也就是整数的最后一位数字，结果为 3。然后通过赋值语句“*num*=*num* / 10；”把 *num* 除以 10 的结果，也就是该整数右移一位的结果 196 赋给变量 *num*（整数 / 整数的运算会自动舍弃小数点之后的部分）。



由于 `num` 的值 196 大于 0，因此会再次执行循环体。接下来显示 196 除以 10 的余数，也就是 1963 倒数第二位的数字 6。同样的处理会在 `num` 大于 0 的时候循环执行，这样就完成了逆向显示的整个过程。

当 `num` 的值为 0 的时候，控制表达式 `num > 0` 就不再成立了（判断的结果为 0），`do` 语句也就结束了。

计算整数的位数

代码清单 4-2 的第二个 `do` 语句的循环次数和输入整数的位数是相同的。这样，只需要统计一下循环的次数，就能得出整数的位数了。

显示输入整数的位数的程序如代码清单 4-3 所示。

代码清单 4-3

```
/*
 * 计算输入的非负整数的位数
 */
#include <stdio.h>

int main(void)
{
    int num;
    int dig;          /* 位数 */

    do {
        printf("请输入一个非负整数:");
        scanf("%d", &num);
        if (num < 0);
        puts("\a 请不要输入负整数。");
    } while (num < 0);          /* 当 num 大于等于 0 时 */

    dig = 0;
    do {
        num = num / 10;        /* 右移一位 */
        dig = dig + 1;
    } while (num > 0);
    printf("该整数的位数是%d.\n", dig);

    return (0);
}
```

运行结果

请输入一个非负整数: -17

请不要输入负整数。

请输入一个非负整数: 1963

该整数的位数是 4。

下面我们还像上一个例子那样，把 1963 的值保存到 `num` 中，分析一下程序的处理流程。首先把 “`num = num/10;`” 的计算结果 196 赋给 `num`，然后通过 “`dig = dig + 1;`” 使 `dig` 的值增加为 1。

该操作会一直循环执行，直到 `num` 的值为 0，这样 `dig` 的值就是 `do` 语句执行的次数，也就是该整数的位数。

num	dig
1963	0
↓	↓
196	1
↓	↓
19	2
↓	↓
1	3
↓	↓
0	4

初始化

让我们思考一下下面的问题。

计算并显示从 1 到 5 的和。

程序如代码清单 4-4 所示。

代码清单 4-4

```
/*
 * 计算并显示从 1 到 5 的和
 */
#include <stdio.h>

int main(void)
{
    int no = 1;
    int sum = 0;

    do {
        sum = sum + no;
        no = no + 1;
    } while (no <= 5);

    printf("1 到 5 的合计值是 %d。\\n", sum);

    return (0);
}
```

运行结果

1 到 5 的合计值是 15。

本例中变量 `no` 和 `sum` 的声明方式跟以前不同，在声明的同时还给它们赋了值。像这样在创建对象（变量）的同时，为其指定初始值的操作称为**初始化**（initialization）。声明中等号右边的常量被称为**初始值**（initializer）（图 4-3）。

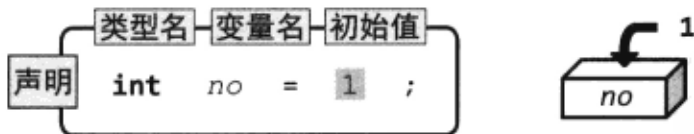


图 4-3 带有初始值的声明

在程序中，通过对 `no` 和 `sum` 的声明，创建出了这两个变量，同时还给它们赋了初始值 0 和 1。

之前我们把变量比作放置整数的盒子，如果我们已经知道了其中应该存放的数值，那么我们自然可以在制作这个盒子的同时把它也放进去了。

■ 注意 ■

在声明变量的时候，除了有特别的要求之外，一定要对其进行初始化。

让我们参考右图来梳理一下程序的流程。

`no` 的初始值是 1, `sum` 的初始值是 0。首先把 `sum = sum + no;` 的结果, 也就是 1 赋给 `sum`。接下来通过语句 `no = no + 1;` 使 `no` 的值增加为 2。此时, `no` 的值仍然小于等于 5, 因此会再次执行循环体中的语句。

然后把 `sum = sum + no;` 也就是 1+2 的结果 3 赋给 `sum`, 再通过语句 `no = no + 1;` 使 `no` 的值增加为 3。这时 `no` 的值仍然小于等于 5, 因此会再次执行循环体中的语句。

循环处理会在 `no` 小于等于 5 的时候反复执行, `no` 的值也就像 1, 2……这样逐渐增加, 直到执行完 5 次循环为止。

在执行循环期间, `sum` 的值也随着 `no` 的不断增长而变化, 记录下了从 1 到 5 的合计值。

另外大家需要注意的是当 **do** 语句结束的时候, `no` 的值并不是 5 而是已经增加到了 6。

► 关于初始化以及对象的生命周期等内容将在第 6 章进行详细说明。

<code>no</code>		<code>sum</code>
1		0
2	0+1→	1
↓		↓
3	1+2→	3
↓		↓
4	3+3→	6
↓		↓
5	6+4→	10
↓		↓
6	10+5→	15

● 练习 4-1

对代码清单 4-2 中的程序进行修改, 像右面这样在显示结果的同时显示出输入的整数值。

请输入一个非负整数: 1963
1963 逆向显示的结果是 3691。

● 练习 4-2

对代码清单 4-3 中的程序进行修改, 在输出结果的同时显示出输入的整数。

请输入一个非负整数: 1963
1963 的位数是 4。

● 练习 4-3

编写一段程序, 像右面这样读取两个整数的值, 然后计算出它们之间所有整数的和。

请输入两个整数。
整数1: 37
整数2: 28
大于等于 28 小于等于 37 的所有整数的和是 325。

复合赋值运算符

接下来我们使用复合赋值运算符（compound assignment operator）对上例的程序进行简化，见代码清单 4-5。

代码清单 4-5

```
/*
 * 计算并显示从 1 到 5 的和（使用复合赋值运算符）
 */

#include <stdio.h>

int main(void)
{
    int no = 1;
    int sum = 0;

    do {
        sum += no;      /* 给 sum 的值加上 no */
        no  += 1;      /* 给 no 的值加上 1 */
    } while (no <= 5);

    printf("1 到 5 的合计值是 %d.\n", sum);

    return (0);
}
```

运行结果

1 到 5 的合计值是 15。

使用了复合赋值运算符的表达式 `sum += no`，表示给 `sum` 的值加上 `no`，也就是 `sum` 的值增加了 `no`。这样的表达式比之前的 `sum = sum + no` 更加直接，而且 `sum` 只需要书写一次（避免了书写错误的发生）。当然 `no+=1` 也是同样的道理，表示 `no` 的值增加了 1。

对于 `*`、`/`、`%`、`+`、`-`、`<<`、`>>`、`&`、`^`、`|` 这些运算符来说，`a @= b` 和 `a = a @ b`^① 的效果是一样的。

运算和赋值可以一举两得的复合赋值运算符如表 4-2 所示。

■ 表 4-2 复合赋值运算符

复合赋值运算符	<code>*=</code>	<code>/=</code>	<code>+=</code>	<code>-=</code>	<code><<=</code>	<code>>>=</code>	<code>&=</code>	<code>^=</code>	<code> =</code>
---------	-----------------	-----------------	-----------------	-----------------	------------------------	------------------------	---------------------	-----------------	-----------------

► 运算符 `<<`、`>>`、`&`、`^`、`|` 将在第 7 章进行讲解。

① 这里的 `@` 指代前面提到的各种运算符。

后置递增运算符和后置递减运算符

接下来让我们使用能够使变量的值加 1 的后置递增运算符 (postfixed increment operator) `++` 来进一步简化程序代码 (见代码清单 4-6)。

代码清单 4-6

```
/*
 * 计算并显示从 1 到 5 的和 (使用复合赋值运算符和后置递增运算符)
 */

#include <stdio.h>

int main(void)
{
    int no = 1;
    int sum = 0;

    do {
        sum += no;          /* 给 sum 的值加上 no */
        no++;              /* no 的值递增 (增加 1) */
    } while (no <= 5);

    printf("1 到 5 的合计值是 %d。\\n", sum);

    return (0);
}
```

运行结果

1 到 5 的合计值是 15。

使用了该运算符的表达式 `no++`, 能够使操作数的值增加 1 (这样的运算通常称为递增)。

当然还存在能够使操作数的值减 1 (递减) 的后置递减运算符 (postfixed decrement operator) `--` (表 4-3)。

表 4-3 后置递增运算符和后置递减运算符

后置递增运算符	<code>a++</code>	使 <code>a</code> 的值增加 1 (该表达式的值是增加前的值)
后置递减运算符	<code>a--</code>	使 <code>a</code> 的值减少 1 (该表达式的值是减小前的值)

► 之后会给大家介绍后置递减运算符的使用实例。

● 练习 4-4

使用复合赋值运算符 `/=`, 对代码清单 4-2 中的程序进行修改。

● 练习 4-5

使用复合赋值运算符 `/=` 和后置递增运算符 `++`, 对代码清单 4-3 中的程序进行修改。

4-2 while语句

while 语句

输入一个整数值，显示出从它开始递减到 0 的每一个整数的程序如代码清单 4-7 所示。

代码清单 4-7

```
/*
从输入整数开始倒数到 0
*/

#include <stdio.h>

int main(void)
{
    int no;

    printf("请输入一个整数:");
    scanf("%d", &no);

    while (no >= 0) {
        printf("%d ", no);
        no--;          /*no的值递减(减少1)*/
    }
    putchar('\n');     /*换行*/

    return (0);
}
```

运行结果 (1)

请输入一个整数: 11
11 10 9 8 7 6 5 4 3 2 1 0

运行结果 (2)

请输入一个整数: 0
0

运行结果 (3)

请输入一个整数: -5

本例中使用的语句是 **while** 语句 (while statement)，它的结构如图 4-4 所示。

while语句



图 4-4 while 语句的结构图

while 语句会在表达式的值达到 0 之前循环执行其中的语句。程序的流程如图 4-5 所示。

while (表达式) 语句



图 4-5 while 语句的流程

► 与上一节介绍的 **do** 语句不同，**while** 语句会在开始的时候判断控制表达式的值 (**do** 语句则在执行完一次循环体内语句之后再判断控制表达式的值)。**do** 语句一定会执行一次循环体中的语句，而 **while** 语句并不一定会执

行循环体中的语句。如果控制表达式的第一次判断结果为 0，则循环体中的语句一次都不会执行。

我们以 `no` 的值等于 11 为例，分析一下程序的运行过程。首先对控制表达式 `no >= 0` 的值进行判断，结果为 1，不为 0，所以循环体中的语句会被执行。先通过 `printf("%d ", no);` 语句在屏幕上显示出 11（11 后面跟着一个空格）。接下来执行 `no--;` 语句，由于后置递减运算符的作用 `no` 的值递减为 10。

然后再对循环是否继续执行进行判断。由于表达式 `no >= 0` 仍然成立（判断结果为 1），因此屏幕上会显示出 10，并且 `no` 的值递减为 9。就这样，通过循环的往复执行，相应的数字会逐一显示在屏幕上。

当 `no` 的值为 0 的时候，在屏幕上显示出 0，接下来通过后置递减运算符使 `no` 的值递减为 -1。这之后循环是否继续执行的控制表达式 `no >= 0` 就不再成立了（判断结果为 0），循环结束。

需要注意的是，虽然最后显示在屏幕上的 `no` 的值是 0，但其实 **while** 语句结束的时候，它的值已经变成了 -1。

字符常量

while 语句结束之后会执行右图所示的语句。像 `'\n'` 和 `'A'` 这样用单引号括起来的字符称为**字符常量**（character constant）。
`putchar('\n');`

► 字符常量是 **int** 类型。像 `'ab'` 这样在 " 中写入多个字符也是可以的，但是需要编译器支持，所以还是希望大家尽量不要使用（由于 `\n` 和 `\a` 这样的转义符是作为一个字符来使用的，因此没有关系）。

putchar

putchar 函数可以用来显示字符。它只有一个参数，就是需要显示的字符。本例中的参数是 `'\n'`，也就是执行了换行操作。

*

输入的值为 0 的时候，**while** 语句只执行一次，在屏幕上显示出 0。输入的值不为负数的时候，**while** 语句不会被执行，而只是进行换行操作。

● 练习 4-6

对代码清单 4-7 中的程序进行修改，当输入值为负数的时候不执行换行操作。

用递减运算符简化程序代码

下面让我们灵活运用后置递减运算符的特性进一步简化倒计数的程序。

代码清单 4-8

```
/*
从输入的整数开始倒数到 0（第二版）
*/

#include <stdio.h>

int main(void)
{
    int no;

    printf("请输入一个正整数:");
    scanf("%d", &no);

    while (no >= 0)
        printf("%d ", no--);           /*no 的值在显示之后递减*/

    putchar('\n');

    return (0);
}
```

运行结果

请输入一个正整数: 11

11 10 9 8 7 6 5 4 3 2 1 0

让我们再仔细看一下表 4-3 中关于后置递增运算符和后置递减运算符的介绍。其中对于 `a--` 的说明是：使 `a` 的值减小 1（该表达式的值是减小前的值）。也就是说对表达式 `a--` 进行判定的时候，它还是递减之前的值。例如，当 `no` 的值是 11 的时候，表达式 `no--` 的结果还是 `no` 的值 11，而不是 10。

因此，显示 `no--` 的值的时候会按照如下步骤执行：

- (a) 显示 `no` 的值。
- (b) 然后对 `no` 的值进行递减操作。

这样就可以进一步简化程序代码了。

● 练习 4-7

对代码清单 4-8 的程序进行修改，使其不是递减到 0 而是递减到 1 为止。

数据递增

这次我们来编写一段跟之前的程序相反的程序，显示出从 0 开始递增到输入整数的各个整数。程序如代码清单 4-9 所示。

代码清单 4-9

```
/*
    递增显示从 0 到输入的正整数为止的各个整数
*/

#include <stdio.h>

int main(void)
{
    int i, no;

    printf("请输入一个正整数:");
    scanf("%d", &no);

    i = 0;
    while (i <= no)
        printf("%d ", i++);          /*i 的值在显示之后递增*/
    putchar('\n');

    return (0);
}
```

运行结果

请输入一个正整数: 12

0 1 2 3 4 5 6 7 8 9 10 11 12

该程序与之前实现递减的程序最大的不同就是引入了一个新的变量 *i*。*i* 的值按照 0, 1, 2, ……逐渐递增。当显示出与 *no* 相同的数值之后，**while** 语句的循环结束。由于显示之后 *i* 的值又递增了一次，因此 **while** 语句结束的时候，*i* 的值比 *no* 的值大 1。

● 练习 4-8

- 对代码清单 4-9 的程序进行如下修改。
- 从 1 开始递增。
 - 输入的值小于 0 的时候不换行。

● 练习 4-9

编写一段程序，像右面这样按照升序显示出小于输入值的所有正偶数。

请输入一个整数: 19

2 4 6 8 10 12 14 16 18

● 练习 4-10

编写一段程序，像右面这样显示出小于输入整数的所有 2 的乘方。

请输入一个整数: 19

2 4 8 16

限定次数的循环操作

输入一个整数后，并排连续显示出该整数个 *****，具体的程序如代码清单 4-10 所示。

代码清单 4-10

```
/*
 * 输入一个整数，连续显示出该整数个 *
 */
#include <stdio.h>

int main(void)
{
    int no;

    printf("请输入一个正整数:");
    scanf("%d", &no);

    while (no-- > 0)
        putchar('*');
    putchar('\n');

    return (0);
}
```

运行结果

请输入一个正整数: 15

让我们以 *no* 的值等于 15 为例来考虑一下。首先，对控制表达式 *no-- > 0* 进行判断。最初表达式 *no > 0* 的判定结果是 1，然后 *no* 的值递减为 14。随着循环的进行，*no* 的值逐渐递减。当它的值递减为 0 的时候，表达式 *no-- > 0* 的值第一次为 0，**while** 语句也就结束了。

于是，**scanf** 函数读取的数值是几，就进行几次循环，这样就完成了显示相同个数 ***** 的操作。

最后一次对表达式进行判定之后，*no* 的值再次递减，因此当 **while** 循环结束的时候，*no* 的值已经变成了 -1。

● 练习 4-11

编写一段程序，像右面这样读取一个整数，并纵向显示出读取到的整数个 *****。如果输入的是 0 以下的整数，则不显示任何内容。

请输入一个整数: 3

*
*
*

前置递增运算符和前置递减运算符

请大家阅读一下代码清单 4-11 中的程序。首先输入一个整数，然后再依次输入该整数个整数，显示出它们的合计值和平均值。

代码清单 4-11

```
/*
    输入规定个数个整数并显示出它们的合计值和平均值
*/

#include <stdio.h>

int main(void)
{
    int i = 0;
    int sum = 0;           /* 合计值 */
    int num, tmp;

    printf("要输入多少个整数:");
    scanf("%d", &num);

    while (i < num) {
        printf("No.%d:", ++i);
        scanf("%d", &tmp);
        sum += tmp;
    }

    printf("合计值: %d\n", sum);
    printf("平均值: %.2f\n", (double)sum / num);

    return (0);
}
```

运行结果

要输入多少个整数: 6

No.1: 65

No.2: 23

No.3: 47

No.4: 9

No.5: 153

No.6: 777

合计值: 1074

平均值: 179.00

本例中使用了表 4-4 中介绍的前置递增运算符（prefixed increment operator）。当然也存在与之对应的前置递减运算符（prefixed decrement operator）。

表 4-4 前置递增运算符和前置递减运算符

前置递增运算符	++a	使a的值增加1（该表达式的值是增加后的值）
前置递减运算符	--a	使a的值减少1（该表达式的值是减少后的值）

前置递增运算符的作用和后置递增运算符一样，都能使操作数自动增长。不过增长的时间点却有所不同。++i 的显示过程如下：

- (a) i 的值递增 1。
 - (b) 显示 i 的值。
- 因此，最初显示的 i 是 1。

4-3 for语句

for 语句

下面我们使用 **for** 语句 (for statement) 对代码清单 4-9 中的递增程序进行修改, 详见代码清单 4-12。

代码清单 4-12

```

/*
   从 0 递增显示到输入的正整数为止 (使用 for 语句)
*/

#include <stdio.h>

int main(void)
{
    int i, no;

    printf("请输入一个正整数:");
    scanf("%d", &no);

    for (i = 0; i <= no; i++)
        printf("%d ", i);
    putchar('\n');

    return (0);
}

```

运行结果

请输入一个正整数: 12

0 1 2 3 4 5 6 7 8 9 10 11 12

乍看上去, 使用 **for** 语句的程序比使用 **while** 语句的程序更简洁。**for** 语句的结构如图 4-6 所示, **for** 后面的括号中存在由分号隔开的三个表达式。

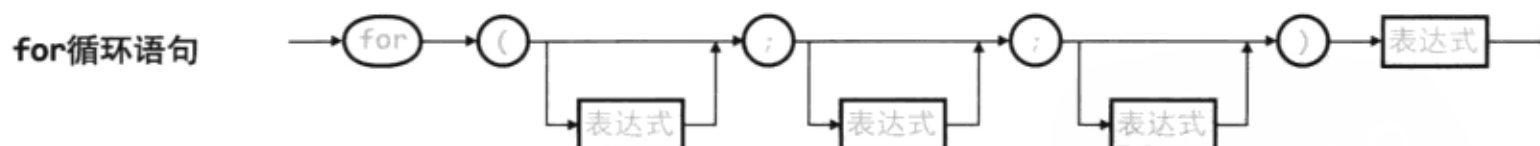


图 4-6 for 语句的结构

for 语句的流程如图 4-7 所示。

for (表达式₁; 表达式₂; 表达式₃) 语句

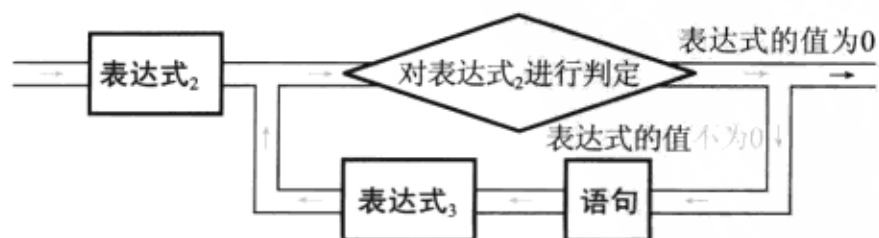


图 4-7 for 语句的流程

接下来请大家看图 4-8。

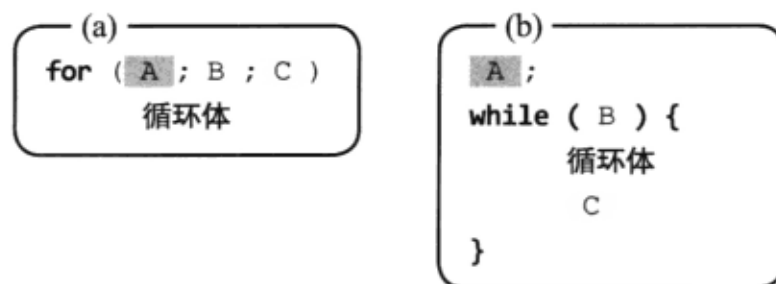


图 4-8 for 语句和 while 语句

首先来看一下图 (a) 中的 **for** 语句。

在循环开始之前，首先会执行预处理 **A**。循环操作会在控制表达式 **B** 的值不为 0 时重复执行。循环体中的语句执行之后，再执行表达式 **C**。

因此，图 (a) 中显示的 **for** 语句和图 (b) 中显示的 **while** 语句效果是相同的。所有的 **for** 语句都可以用 **while** 语句进行替换，所有的 **while** 语句也可以用 **for** 语句进行替换。

下面给大家介绍一下 **for** 语句中需要注意的细节。

A 预处理

表达式 **A** 仅在循环执行之前执行一次。

当程序无需预处理的时候，表达式 **A** 可以省略。

B 控制表达式

表达式 **B** 是用来判定循环操作是否继续执行的表达式。如果该表达式的值不为 0，则执行循环体中的语句。

当省略表达式 **B** 的时候，通常认为控制表达式的值始终不为 0。因此，除非使用 5-3 节中介绍的 **break** 语句，否则该循环将成为永远执行的无限循环。

C 收尾处理

表达式 **C** 会在循环体最后执行。如果没有需要执行的内容，则该表达式可以省略。

循环语句

本章中介绍的 **do** 语句、**while** 语句和 **for** 语句都是用来执行循环操作的语句，因此统称为循环语句（iteration statement）。

固定次数的循环

使用 **for** 语句对代码清单 4-10 中的程序进行修改，实现同样的功能（输入一个整数，并排连续显示出该整数个 *****）。程序如代码清单 4-13 所示。

代码清单 4-13

```
/*
 * 输入一个整数，连续显示出该整数个 * (使用 for 语句)
 */
#include <stdio.h>

int main(void)
{
    int i, no;

    printf("请输入一个正整数: ");
    scanf("%d", &no);

    for (i = 1; i <= no; i++)
        putchar('*');
    putchar('\n');

    return (0);
}
```

运行结果

请输入一个正整数: 12

本程序中的 **for** 语句也可以替换为如下形式。需要注意的是，*i* 的初始值不同（0 或 1），控制表达式所使用的运算符也不相同（**<** 和 **<=**）。

```
for (i = 0; i < no; i++)
    putchar('*');
```

使用了 **while** 语句的代码清单 4-10 中的程序，在 **while** 语句结束的时候，变量 *no* 的值递减成了 -1。

```
while (no-- > 0)
    putchar('*');
```

图 4-9 所示的 **while** 语句和 **for** 语句都执行了 *no* 次循环操作。

for (<i>i</i> = 0; <i>i</i> < <i>no</i> ; <i>i</i> ++)	while (<i>no</i> -- > 0)
语句	语句
for (<i>i</i> = 1; <i>i</i> <= <i>no</i> ; <i>i</i> ++)	while (-- <i>no</i> >= 0)
语句	语句

图 4-9 执行 *no* 次循环的 **for** 语句和 **while** 语句

使用 **for** 语句对代码清单 4-11 中的程序进行修改，实现同样的功能（输入规定个数个整数并显示出它们的合计值和平均值）。修改后的程序如代码清单 4-14 所示。

代码清单 4-14

```
/*
 输入规定个数个整数并显示出它们的合计值和平均值（使用 for 语句）
*/

#include <stdio.h>

int main(void)
{
    int i;
    int sum = 0;           /* 合计值 */
    int num, tmp;

    printf("输入多少个整数: ");
    scanf("%d", &num);

    for (i = 0; i < num; i++) {
        printf("No.%d:", i + 1);
        scanf("%d", &tmp);
        sum += tmp;
    }

    printf("合计值: %d\n", sum);
    printf("平均值: %.2f\n", (double)sum / num);

    return (0);
}
```

运行结果

输入多少个整数: 6

No.1: 65

No.2: 23

No.3: 47

No.4: 9

No.5: 153

No.6: 777

合计值: 1074

平均值: 179.00

● 练习 4-12

使用 **for** 语句对代码清单 4-6 中的程序进行修改，计算 1 到 5 的和。

● 练习 4-13

编写一段程序，像右面这样根据输入整数后，循环显示 1234567890，显示的位数和输入的整数值相同。

请输入一个整数: 25

1234567890123456789012345

● 练习 4-14

编写一段程序，像右面这样显示出身高和标准体重的对照表。显示的身高范围和间隔由输入的整数值进行控制，标准体重精确到小数点后 2 位。

开始数值 (cm): 150

结束数值 (cm): 190

间隔数值 (cm): 5

150cm 45.00kg

155cm 49.50kg

... (以下省略) ...

4-4 多重循环

九九乘法表

使用 **for** 语句显示九九乘法表的程序如代码清单 4-15 所示。

代码清单 4-15

```
/*
 * 显示九九乘法表
 */
#include <stdio.h>

int main(void)
{
    int i, j;

    for (i = 1; i <= 9; i++) {
        for (j = 1; j <= 9; j++)
            printf("%3d", i * j);
        putchar('\n');
    }

    return (0);
}
```

运行结果

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

本程序中使用了嵌套的 **for** 语句，也就是在 *i* 从 1 到 9 递增的 **for** 语句中，嵌套了 *j* 从 1 递增到 9 的 **for** 语句。

它们结合起来执行了如下处理。

当 <i>i</i> 为 1 的时候	执行 <i>j</i> 从 1 递增到 9 的循环操作
当 <i>i</i> 为 2 的时候	执行 <i>j</i> 从 1 递增到 9 的循环操作
⋮	⋮
当 <i>i</i> 为 9 的时候	执行 <i>j</i> 从 1 递增到 9 的循环操作

这样就完成了从 1×1 到 9×9，共计 81 个数值的输出。

● 练习 4-15

编写一段程序，像右面这样为九九乘法表增加横纵标题。

		1	2	3	4	5	6	7	8	9
1		1	2	3	4	5	6	7	8	9
2		2	4	6	8	10	12	14	16	18
3		3	6	9	12	15	18	21	24	27
4		4	8	12	16	20	24	28	32	36

(以下省略)

多重循环

在显示九九乘法表的程序中，我们把一个 **for** 语句嵌套进了另一个 **for** 语句之中。其实，本章中介绍的 **do** 语句、**while** 语句和 **for** 语句都可以任意进行嵌套。

像这样在一个循环语句中嵌套另一个循环语句的循环称为**二重循环**。与之对应的还有三重循环、四重循环等。它们统称为**多重循环**。

请大家看一下代码清单 4-16 中的程序。

代码清单 4-16

```
/*
 * 输入一个非负整数，连续显示出该非负整数个 *（循环次数可任意指定）
 */

#include <stdio.h>

int main(void)
{
    int cont;

    do {
        int num, i;

        do {
            printf("请输入一个非负整数:");
            scanf("%d",&num);
            if (num < 0)
                puts("\a 请不要输入负整数。");
        } while (num < 0);

        for (i = 1; i <= num; i++)
            putchar('*');
        putchar('\n');

        printf("是否继续执行? 【Yes...0/No...9】:");
        scanf("%d",&cont);
    } while (!cont);

    return (0);
}
```

运行结果

```
请输入一个非负整数: 17
*****
是否继续执行? 【Yes...0/No...9】: 0
请输入一个非负整数: 5
*****
是否继续执行? 【Yes...0/No...9】: 9
```

本程序可以看作是代码清单 4-1（按照意愿重复执行处理的 **do** 语句）、代码清单 4-2（读取非负整数的 **do** 语句）和代码清单 4-13（连续显示 * 的 **for** 语句）的集合。

在 **do** 语句中分别嵌套了 **do** 语句和 **for** 语句。

长方形

通过 * 的横竖排列显示出一个长方形的程序如代码清单 4-17 所示。

代码清单 4-17

```
/*
 * 描绘一个长方形
 */
#include <stdio.h>

int main(void)
{
    int i, j;
    int width; height;

    puts("让我们来描绘一个长方形。");
    printf("宽: ");      scanf("%d", &width);
    printf("高: ");      scanf("%d", &height);

    for (i = 1; i <= height; i++) {
        for (j = 1; j <= width; j++)
            putchar('*');
        putchar('\n');
    }

    return (0);
}
```

运行结果

让我们来描绘一个长方形。

宽: 5

高: 3

/* 长方形有 height 行 */

/* 显示 width 个 '*' */

/* 换行 */

共计 height 行，每一行都显示出 width 个 *，这样就形成了一个长方形。

直角三角形

通过 * 的排列描绘一个直角三角形的程序如代码清单 4-18 和代码清单 4-19 所示。两段程序的外层循环都像下面这样，通过变量 *ln* 来控制直角三角形的层数。

```
for (i = 0; i <= ln; i++)
```

代码清单 4-19 中的程序(直角在右下方)更复杂一些，在一个 **for** 语句中嵌套了两个 **for** 循环。

● 练习 4-16

对代码清单 4-17 中的程序进行修改，显示出一个横向较长的长方形。

让我们来画一个长方形。

一边: 3

另一边: 7

● 练习 4-17

对代码清单 4-18 和代码清单 4-19 中的程序进行修改，分别显示出直角在左上方和右上方的直角三角形。

代码清单 4-18

```

/*
 * 显示一个直角在左下方的直角三角形
 */

#include <stdio.h>

int main(void)
{
    int i, j, ln;

    printf("三角形有几层: ");
    scanf("%d", &ln);

    for (i = 1; i <= ln; i++) {
        for (j = 1; j <= i; j++)
            putchar('*');
        putchar('\n');
    }

    return (0);
}

```

运行结果

三角形有几层: 5

```

*
**
***
****
*****

```

代码清单 4-19

```

/*
 * 显示一个直角在右下方的直角三角形
 */

#include <stdio.h>

int main(void)
{
    int i, j, ln;

    printf("三角形有几层: ");
    scanf("%d", &ln);

    for (i = 1; i <= ln; i++) {
        for (j = 1; j <= ln - i; j++)
            putchar(' ');
        for (j = 1; j <= i; j++)
            putchar('*');
        putchar('\n');
    }

    return (0);
}

```

运行结果

三角形有几层: 5

```

*
 **
  ***
   ****
    *****

```

● 练习 4-18

编写一段程序，输入一个整数，像右面这样显示出输入整数层的金字塔形状。

让我们来描绘一个金字塔。
金字塔有几层: 3

```

*
 ***
*****

```

4-5 程序的组成元素和格式

关键字

在 C 语言中，像 **if** 和 **else** 这样的标识符被赋予了特殊的意义。这种具有特殊意义的标识符称为**关键字** (keyword)，它们是不能用作变量名的。C 语言的 32 个关键字如下所示^①。

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while				

► 另外，在有些编译器中，asm 和 fortran 也被当作关键字来使用。

标识符

标识符 (identifier) 是赋予给程序中变量和函数 (第 6 章将会学习函数相关的知识) 的名称 (专题 4-1)。

标识符的构成必须像图 4-10 所示的那样。

也就是必须以**非数字**开头，之后可以是非数字和数字的组合。这里的非数字包括大小写字母和下划线。同时，C 语言区分大小写，ABC、abc 和 aBc 分别代表不同的标识符。

合法的标识符示例如下所示：

```
a x1 __y abc_def max_of_group xyz Ax3 If iF IF if3
```

非法的标识符示例如下所示：

```
if 123 98pc
```

► 以下划线开头的标识符 `_x`、`_sin` 和单独的大写英文字母标识符，有可能是编译器内部使用的，因此最好不要用做变量和函数的标识符。

^① C99 标准中又加入了 inline、restrict、_Bool、_Complex 和 _Imaginary 等关键字。

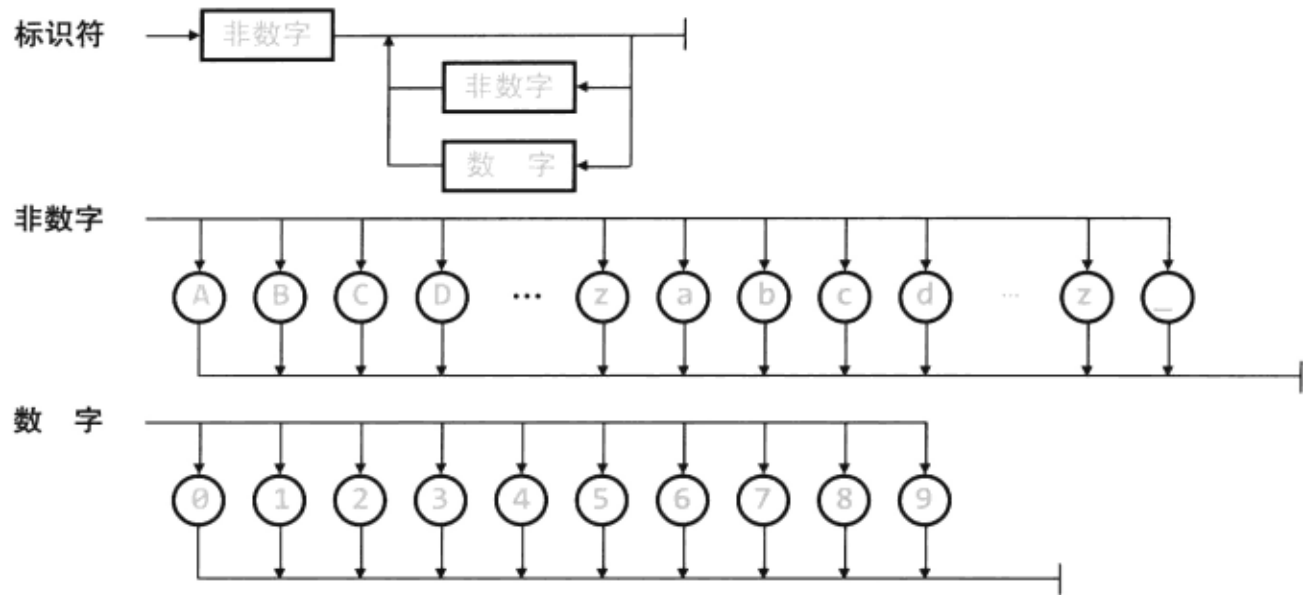


图 4-10 标识符·非数字·数字的结构图

分隔符

关键字和标识符都可以理解为构成语句的单位，用来分隔这些单位的符号就是分隔符 (punctuator)。分隔符一共有 13 种 ([、]、(、)、{、}、*、/、:、=、;、...、#)。

运算符

目前为止我们已经介绍了 + 和 - 等运算符 (operator)。所有运算符的一览表请参考 7-4 节。

常量和字符串常量

字符常量、整数常量和字符串常量都是程序的构成要素。

专题 4-1 姓名和标识符

顾名思义，“标识符”就是为了用来和其他字符进行区分的。在那些讲述未来世界的电影中，人类都被分配了唯一的 ID 号码，每个人的 ID 都不会与其他人重复。

所谓的“姓名”也是如此，是分配给每个人的。不过它并不能保证每个人都使用不同的名字，也就是说存在同名同姓的可能。

如果程序中也存在同名同姓的变量将会是件十分麻烦的事情。因此使用专门的“标识符”就是一个非常理想的解决方案。

另外，大家还可以参考 12-1 节关于命名空间的介绍。

自由的书写格式

下述程序和本书开篇代码清单 1-1 中的程序本质上是一样的，显示的运行结果也一样。

```
/*
 * 显示出整数 15 和 37 的和
 */
#include <stdio.h>

int main(
void)
{ printf("%d", 15 +
37); /* 以十进制数的形式显示
整数 15 和 37 的和 */ return (
0); }
```

运行结果

52

C 语言原则上允许开发人员以自由的格式编写程序。它并不像有些编程语言那样，规定了程序中必须从第几个字符开始写，或者每条语句必须写在一行之内等。

上述程序就是一个自由书写的例子。不过再怎么自由也还是有一些限制的。

(1) 构成语句的单位中间不能插入空格类字符

例如 **int** 和 **return** 这样的关键字，变量 *n1* 和 *n2* 这样的标识符，都是构成语句的单位。在它们中间是不能插入空格类字符的（空格、制表符、换行等）。如下的书写格式是不允许的。

```
ret
    urn(0);
```

(2) 预处理指令中间不能换行

允许使用自由书写格式的 C 语言中也对 **#include** 这样以 **#** 开头的预处理指令有特殊要求。原则上这些指令都必须写在一行内（也存在多行书写的方法）。下面这样的方式是不允许的。

```
#include
    <stdio.h>
```

预处理指令除了 **#include** 指令外，还有 **#define** 指令（5-1 节）等。

(3) 字符串常量和字符常量中间不能换行

用双引号括起来的字符串常量 "..." 也是构成语句的单位，因此也不能像下面这样在中间进行换行。

```
puts (" 在很久很久以前有个地方住着一位老公公和一位老婆婆。
```

```
老公公深深地爱着老婆婆。");
```

连接相邻的字符串常量

在源代码中对字符串常量进行分割，然后用双引号把每个部分括起来，这样就可以漂亮地书写较长的字符串常量了。例如我们可以像下面这样书写程序：

```
puts (" 在很久很久以前有个地方住着一位老公公和一位老婆婆。" /* 在下一行继续书写 */
      "老公公深深地爱着老婆婆。");
```

可以把被空格类字符以及注释分隔开的相邻字符串常量作为一个整体来看待。

缩进

请大家回顾一下代码清单 4-16 中的程序。在程序的每一行开头都有 4 位空白。复合语句 { } 中包含一系列的声明和语句，就像我们常说的“段落”一样。根据经验，在段落中统一向右移几位进行书写，可以更容易地理解程序结构，更方便阅读。像这样以段落为单位向右移动的书写方式称为**缩进**（也称为“分段处理”）。本书中的程序全部使用 4 位缩进。

另外，在运算符的前后加入空格也可以使表达式不那么拥挤，方便阅读。

专题 4-2 do 语句和复合语句

请大家看一下左下方的程序。这里的 `while(a < 8)` 是 `do` 语句的后半部分，而不是 `while` 语句的一部分。但是乍看上去，有可能会误认为这是 `while` 语句。因此，在书写 `do` 语句的时候，为了让循环语句更加清晰，就需要像右下方的程序那样用大括号把复合语句括起来。这样就可以一目了然地看出它不是 `while` 语句了。

```
do
    a = a + 1;
while (a < 8);
if (b == 5) c = 3;
```

```
do {
    a = a + 1;
} while (a < 8);
if (b == 5) c = 3;
```

11

12

13

14

15

16

17

18

19

20

21

22

23

24

第 5 章

数 组

学生的学籍号码、棒球选手背后的号码，还有飞机的座位号码……在生活中我们经常会遇到相同类型的事物聚集在一起的情况，与其逐一叫出它们的名字，还不如统一使用“号码”更加简单明了。举个例子，对于超过 100 个的飞机座位来说，如果分别称为“鹤座”、“松座”……将会是一种什么样的情形呢？

本章将会为大家介绍为了提高处理效率而把具有相同类型的数据有序地组织起来的一种形式——数组。



5-1 数组

数组

依次输入 5 名学生的分数，显示出他们的总分和平均分。具体程序如代码清单 5-1 所示。

代码清单 5-1

```
/*
    输入 5 名学生的分数并显示出他们的总分和平均分
*/

#include <stdio.h>

int main(void)
{
    int uchida;          /* 内田同学的分数 */
    int satoh;           /* 佐藤同学的分数 */
    int sanaka;          /* 佐中同学的分数 */
    int hiraki;          /* 平木同学的分数 */
    int masaki;          /* 真崎同学的分数 */
    int sum = 0;         /* 总分 */

    puts(" 请输入分数。");
    printf("1 号: "); scanf("%d", &uchida);
    printf("2 号: "); scanf("%d", &satoh);
    printf("3 号: "); scanf("%d", &sanaka);
    printf("4 号: "); scanf("%d", &hiraki);
    printf("5 号: "); scanf("%d", &masaki);

    sum += uchida;
    sum += satoh;
    sum += sanaka;
    sum += hiraki;
    sum += masaki;

    printf(" 总分: %5d\n", sum);
    printf(" 平均分: %5.1f\n", (double)sum / 5);

    return (0);
}
```

运行结果

请输入分数。
1 号: 95
2 号: 83
3 号: 85
4 号: 63
5 号: 89
总分: 415
平均分: 83.0

如果学生的人数不是 5 名而是 300 名的话会怎么样呢？为了保存分数，需要创建 300 个变量，而且还必须管理 300 个变量名。编写程序的时候光是注意不键入错误的变量名就已经很麻烦了。

擅长处理这类数据的就是**数组** (array)，它通过“号码”把相同数据类型的变量集中起来进行管理。

一般的对象声明形式如下所示 (图 5-1a)。

```
int vx;          /* 非数组对象 */
```

与之相对，数组的声明形式如下所示 (图 5-1b)。

```
int vc[5];       /* 数组对象 */
```

通过这样的声明，就创建了一个包含 5 个 **int** 型变量的数组 **vc**。

数组中的各个元素，可以通过在数组名称后面的**下标运算符** (subscript operator) [] 中加入“号码”来表示，如：**vc[0]**、**vc[1]**、**vc[2]**、**vc[3]**、**vc[4]**。

这些号码就是被称为**下标** (subscript) 的整数值。下标从 0 开始，按照 0、1、2、3、……这样的顺序增加。与其说下标代表了数组中某个元素的位置，还不如说是代表了该元素在首个元素后的位置。

► 其他的编程语言里，有些下标从 1 开始，有些可以任意指定下标的初始值，请大家特别注意这些不同之处。

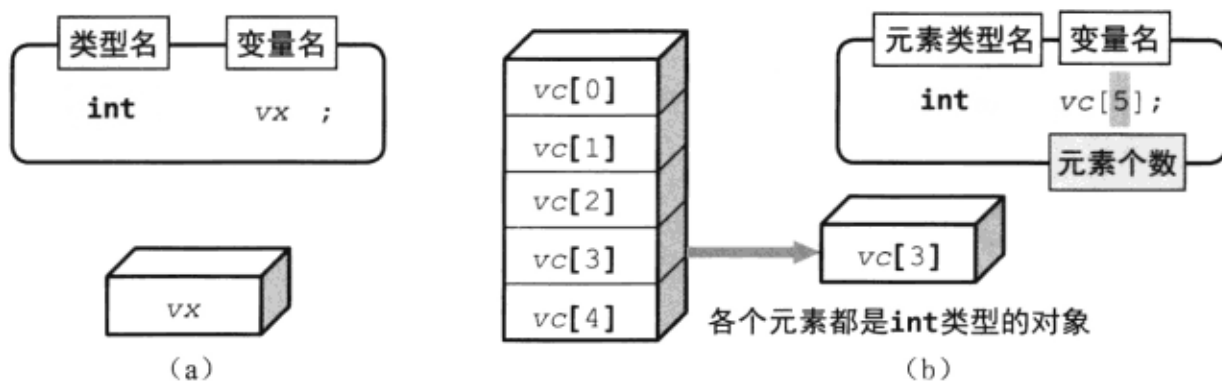


图 5-1 数组

从图中可以看出，数组对象就是同一类型对象的集合。从数组 **vc** 中取得的任何元素都是 **int** 类型的对象。不会出现既存在 **int** 类型，又存在 **double** 类型的情况。另外，构成数组的元素的类型称为**元素类型**。数组 **vc** 的元素类型是 **int** 型。

■ 注意 ■

数组实现了相同类型对象的集合。

► 不同类型的对象组成的集合不是数组而是**结构体**。我们将在第 12 章学习关于结构体的知识。

请大家注意，声明数组的时候，元素个数必须是**常量**。下面这样的声明是会出现错误的^①。

```
int n = 5;
int a[n];          /* 错误：元素个数必须是常量 */
```

① C99支持变长数组。因此下面的声明在C99中是正确的。

数组和 for 语句

创建一个元素类型为 **int**，包含 5 个元素的数组，依次把 1、2、3、4、5 赋给它们并进行显示。程序如代码清单 5-2 所示。

代码清单 5-2

```
/*
    依次把 1、2、3、4、5 赋值给数组每个元素并显示
*/

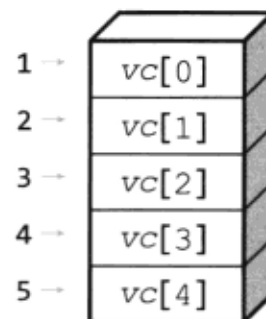
#include <stdio.h>

int main(void)
{
    int vc[5];          /* 包含 5 个元素的数组 */

    vc[0] = 1;
    vc[1] = 2;
    vc[2] = 3;
    vc[3] = 4;
    vc[4] = 5;

    printf("vc[0] = %d\n", vc[0]);
    printf("vc[1] = %d\n", vc[1]);
    printf("vc[2] = %d\n", vc[2]);
    printf("vc[3] = %d\n", vc[3]);
    printf("vc[4] = %d\n", vc[4]);

    return (0);
}
```



运行结果

```
vc[0] = 1
vc[1] = 2
vc[2] = 3
vc[3] = 4
vc[4] = 5
```

使用 **for** 语句对上述程序进行修改后的结果如代码清单 5-3 所示。

代码清单 5-3

```
/*
    依次把 1、2、3、4、5 赋值给数组个元素并显示（使用 for 语句）
*/

#include <stdio.h>

int main(void)
{
    int i;
    int vc[5];          /* 包含 5 个元素的数组 */

    for (i = 0; i < 5; i++)
        vc[i] = i + 1;

    for (i = 0; i < 5; i++)
        printf("vc[%d] = %d\n", i, vc[i]);

    return (0);
}
```

运行结果

```
vc[0] = 1
vc[1] = 2
vc[2] = 3
vc[3] = 4
vc[4] = 5
```

先来看一下为数组元素赋值的第一个 **for** 语句。这个 **for** 语句中的 *i* 从 0 开始递增，一共进行了 5 次循环操作。因此可以分解为以下步骤：

<i>i</i> 为 0 的时候	<code>vc[0] = 0 + 1;</code>	<code>/* vc[0] = 1; */</code>
<i>i</i> 为 1 的时候	<code>vc[1] = 1 + 1;</code>	<code>/* vc[1] = 2; */</code>
<i>i</i> 为 2 的时候	<code>vc[2] = 2 + 1;</code>	<code>/* vc[2] = 3; */</code>
<i>i</i> 为 3 的时候	<code>vc[3] = 3 + 1;</code>	<code>/* vc[3] = 4; */</code>
<i>i</i> 为 4 的时候	<code>vc[4] = 4 + 1;</code>	<code>/* vc[4] = 5; */</code>

这样就完全替代了代码清单 5-2 中的赋值处理。后面负责显示的第二个 **for** 语句也是同样的道理。

对数组进行具有一定规律的操作时，如果能够很好地使用 **for** 语句，就能使程序更加简洁。

*

为 **double** 型数组的全部元素赋值（0.0）的程序如代码清单 5-4 所示。

代码清单 5-4

```

/*
  将数组的全部元素赋值为（0.0）并显示
*/
#include <stdio.h>

int main(void)
{
    int i;
    double vd[5];          /* 包含 5 个元素的数组 */

    for (i = 0; i < 5; i++)
        vd[i] = 0.0;

    for (i = 0; i < 5; i++)
        printf("vd[%d] = %.1f\n", i, vd[i]);

    return (0);
}
    
```

运行结果

```

vd[0] = 0.0
vd[1] = 0.0
vd[2] = 0.0
vd[3] = 0.0
vd[4] = 0.0
    
```

● 练习 5-1

对代码清单 5-3 中的程序进行修改，从头顺次为数组中的元素赋值 0、1、2、3、4。

● 练习 5-2

对代码清单 5-3 中的程序进行修改，从头顺次为数组中的元素赋值 5、4、3、2、1。

数组初始化

上一章给大家介绍了在声明变量的时候，除了的确没有必要的情况，都需要对变量进行初始化。下面我们对代码清单 5-2 和代码清单 5-3 中的程序进行修改，加入对数组元素进行初始化的处理（代码清单 5-5）。

代码清单 5-5

```
/*
  从头开始顺次为数组各元素进行初始化（1、2、3、4、5）并进行显示
*/
#include <stdio.h>

int main(void)
{
    int i;
    int vc[5] = {1, 2, 3, 4, 5};    /* 初始化 */

    for (i = 0; i < 5; i++)
        printf("vc[%d] = %d\n", i, vc[i]);

    return (0);
}
```

运行结果

```
vc[0] = 1
vc[1] = 2
vc[2] = 3
vc[3] = 4
vc[4] = 5
```

数组的初始值就是那些在大括号中的、用逗号分隔并逐一赋给各个元素的值。上述程序按照 `vc[0]`、`vc[1]`、`vc[2]`、`vc[3]`、`vc[4]` 的顺序，使用 1、2、3、4、5 对数组 `vc` 进行了初始化。

还可以像下面这样在声明数组的时候不指定元素个数，数组会根据初始值的个数自动进行设定。

```
int vc[] = {1, 2, 3, 4, 5};
```

另外，数组的初始化还有这样一个原则：当初始值的数量不足时，会自动用 0 对剩余的元素进行初始化。因此，如果想要使用 0 初始化数组中的全部元素，下面这种方式是比较好的选择。赋值语句中并没有给第二个及其后面的元素赋初始值，因此编译器会自动用 0 来初始化它们。

```
int vc[5] = {0};    /* 使用 {0, 0, 0, 0, 0} 进行初始化 */
```

对于下面的赋值语句，编译器会自动使用 0 来初始化第三个及其后面的元素。

```
int vc[5] = {0, 1};    /* 使用 {0, 1, 0, 0, 0} 进行初始化 */
```

► 如下所示，当初始值的个数超过数组的元素个数的时候，程序会发生错误。

```
int vc[3] = {1, 2, 3, 4};    /* 错误：初始值过多 */
```

另外，不能通过赋值语句进行初始化。下面是一个错误的例子。

```
int vc[3];

vc = {1, 2, 3};    /* 错误：不能使用赋值语句进行初始化 */
```

数组的复制

请大家先来看一下代码清单 5-6 中的程序。

代码清单 5-6

```

/*
    把数组中的全部元素复制到另一个数组中
*/

#include <stdio.h>

int main(void)
{
    int i;
    int va[5] = {15, 20, 30};    /* 使用 {15, 20, 30, 0, 0} 进行初始化 */
    int vb[5];

    for (i = 0; i < 5; i++)
        vb[i] = va[i];

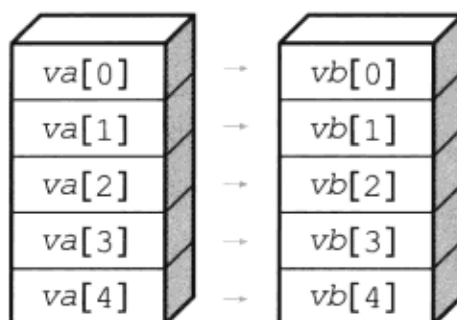
    puts(" va vb");
    puts("-----");
    for (i = 0; i < 5; i++)
        printf("%3d%3d\n", va[i], vb[i]);

    return (0);
}

```

运行结果

va	vb
15	15
20	20
30	30
0	0
0	0



程序中的第一个 **for** 语句，把 `va` 中全部元素的值依次赋给 `vb` 中的元素。第二个 **for** 语句显示出了两个数组中所有元素的值。根据程序的运行结果可以清楚地发现数组已经被正确地复制了。

但是，这样的操作是不是有点繁琐呢。

事实上，C 语言并不支持数组的互相赋值。类似下面这样的语句是错误的。

```
vb = va;    /* 错误：不能为数组赋值 */
```

因此，只能像上述程序那样，对数组的全部元素逐一赋值。

■ 注意 ■

不能使用赋值运算符为数组赋值。

● 练习 5-3

对代码清单 5-5 中的程序进行修改，从头开始依次使用 5、4、3、2、1 进行赋值。

● 练习 5-4

对代码清单 5-6 中的程序进行修改，将数组 `va` 中的元素按照倒序复制到数组 `vb` 中。

输入数组元素的值

逐一输入数组各元素的值并进行显示。程序如代码清单 5-7 所示。

代码清单 5-7

```
/*
    输入数组元素的值并显示
*/

#include <stdio.h>

int main(void)
{
    int i;
    int vx[5];

    for (i = 0; i < 5; i++) {
        printf("vx[%d]:", i);
        scanf("%d", &vx[i]);
    }

    for (i = 0; i < 5; i++)
        printf("vx[%d]=%d\n", i, vx[i]);

    return (0);
}
```

运行结果

```
vx[0] : 17
vx[1] : 38
vx[2] : 52
vx[3] : 41
vx[4] : 63
vx[0] = 17
vx[1] = 38
vx[2] = 52
vx[3] = 41
vx[4] = 63
```

使用 **scanf** 函数存储键盘输入值的方法，与其他（数组以外）变量的情况完全一样。

► 使用 **scanf** 函数读取输入信息的时候，需要在变量前加上 **&** 符。

对数组进行倒序排列

依次顺序读取数组中各元素的值，并按照倒序显示出来。具体程序如代码清单 5-8 所示。

对五个元素进行倒序排列的操作如图 5-2 所示，分为如下

两个步骤：

把 **vx[0]** 和 **vx[4]** 的值进行交换

把 **vx[1]** 和 **vx[3]** 的值进行交换

程序中的第二个 **for** 语句实现了上述功能。

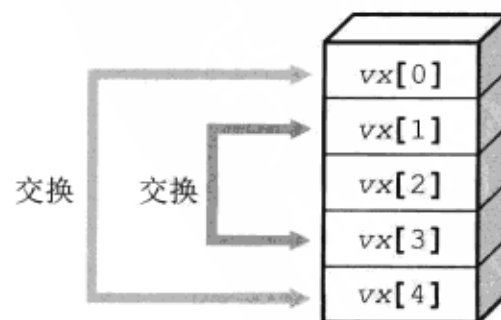


图 5-2 数组排序

代码清单 5-8

```

/*
对数组中的全部元素进行倒序排列
*/

#include <stdio.h>

int main(void)
{
    int i;
    int vx[5];

    for (i = 0; i < 5; i++) {
        printf("vx[%d]:", i);
        scanf("%d", &vx[i]);
    }

    for (i = 0; i < 2; i++) {
        int temp = vx[i];
        vx[i] = vx[4 - i];
        vx[4 - i] = temp;
    }

    for (i = 0; i < 5; i++)
        printf("vx[%d]=%d\n", i, vx[i]);

    return (0);
}

```

运行结果

```

vx[0] : 17
vx[1] : 38
vx[2] : 52
vx[3] : 41
vx[4] : 63
vx[0] = 63
vx[1] = 41
vx[2] = 52
vx[3] = 38
vx[4] = 17

```

第二个 **for** 语句的循环体执行了交换两个整数值的操作。通常情况下，如果想交换两个变量 x 、 y 的值，下面这样的操作是无法实现的（两个变量的值都会变为 y 的初始值）。

```
x = y;
```

```
y = x;
```

为了实现交换操作，必须使用一个额外的变量（图 5-3）。处理流程如下所示。

- (1) 把 x 的值保存在 $temp$ 中。
- (2) 把 y 的值赋给 x 。
- (3) 把 $temp$ 中保存的值赋给 y 。

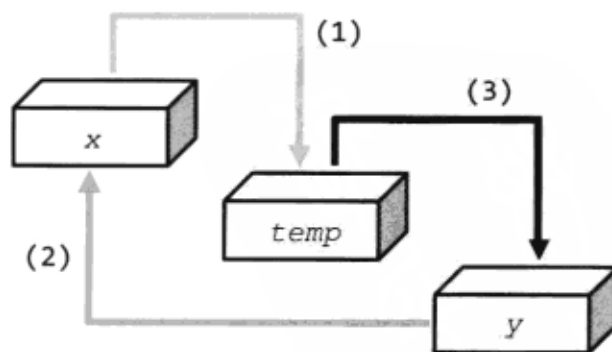


图 5-3 交换两个值

● 练习 5-5

对代码清单 5-8 中的程序进行修改，把数组的元素个数增加为 8 个。

使用数组进行成绩处理

对本章开头代码清单 5-1 中的成绩处理程序进行修改，使用数组完成同样的功能。程序如代码清单 5-9 所示。

代码清单 5-9

```
/*
 输入 5 名学生的分数并显示出他们的总分和平均分
*/

#include <stdio.h>

int main(void)
{
    int i;
    int tensu[5];           /*5 名学生的分数*/
    int sum = 0;            /* 总分 */

    puts("请输入学生的分数。");
    for (i = 0; i < 5; i++) {
        printf("%2d 号:", i + 1);
        scanf("%d", &tensu[i]);
        sum += tensu[i];
    }

    printf("总分: %5d\n", sum);
    printf("平均分: %5.1f\n", (double)sum / 5);

    return (0);
}
```

运行结果

请输入学生的分数。

1 号: 95

2 号: 83

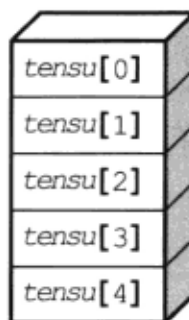
3 号: 85

4 号: 63

5 号: 89

总分: 415

平均分: 83.0



数组 *tensu* 用来保存学生的分数。同时，由于数组的下标是从 0 到 4，因此在提示输入学生分数的时候，要使用下标值加 1（即，* 号:）来进行显示。

让我们来考虑一下学生人数增多，导致数组的元素个数不得不增加为 8 的情况。首先需要把程序中的 5 全都替换成 8。当然，与学生人数无关的数字 5 无需替换，因此这里需要进行选择性替换。本程序中前两个数字 5 和最后一个数字 5 需要替换为 8，后两个 5 无需替换。这个工作还是比较繁琐的。

对象式宏

可以解决上述问题的就是对象式宏（object-like macro）。请大家看一下代码清单 5-10 中的程序。

本程序的关键部分是 **#define 指令**。这里定义的名称 NUMBER 称为宏名称（macro name）。为了与其他变量相区别，宏的名称通常定义为大写字母。

代码清单 5-10

```

/*
    输入 5 名学生的分数并显示出他们的总分和平均分
*/

#include <stdio.h>

#define NUMBER 5          /* 学生人数 */

int main(void)
{
    int i;
    int tensu[NUMBER]; /* 学生的分数 */
    int sum = 0;        /* 总分 */

    puts("请输入学生的分数。");
    for (i = 0; i < NUMBER; i++) {
        printf("%2d 号 :", i + 1);
        scanf("%d", &tensu[i]);
        sum += tensu[i];
    }

    printf("总分: %5d\n", sum);
    printf("平均分: %5.1f\n", (double) sum / NUMBER);

    return (0);
}

```

运行结果

请输入学生的分数。

1 号 : 95 2 号 : 83 3 号 : 85 4 号 : 63 5 号 : 89

总分: 415

平均分: 83.0

大家是否了解文字处理机或者编辑器的替换处理呢？**#define** 指令和它们的原理其实是一样的。程序中所有写了 **NUMBER** 的地方都会被 5 代替，然后再进行编译与执行处理。

当然，如果需要改变人数，只要修改下面这一个地方就可以了。

```
#define NUMBER 8
```

在程序中使用宏，不仅能够在一个地方统一管理，而且通过为常量定义名称，还可以使程序阅读起来更容易。如果能够加上恰当的注释，效果会更加明显。

还有一种观点认为，只要执行了正确的操作就好，即使不使用宏，也可以编写出表面上同等质量的程序。但是，使用了宏的程序，其内在的质量要更高一筹。

■ 注意 ■

不要在程序中直接使用数值，最好能够通过宏的形式定义出它们的名称。定义宏的时候，请不要忘记添加注释。

- ▶ 对象式宏并不能够用来替换字符串字面量和字符常量中的部分内容，也不能用来替换变量名等标识符中的部分内容。

赋值表达式

通常，对于表达式来说，都可以对其值进行判定。赋值表达式也可以进行同样的判定操作。例如像下面这样对 **int** 型变量 *x* 进行赋值的操作。由于整数 *x* 无法保存小数点后面的数值，因此它的值变成了 2。

```
x = 2.95;
```

请大家牢记以下内容。

■ 注意 ■

对赋值语句进行判定的时候，该语句的类型和值与赋值后的左操作数相同。

赋值表达式 *x* = 2.95 的判定结果与赋值后的左操作数，也就是 *x* 的类型和值相同，都是 **int** 类型的 2。

让我们在此基础上来理解一下代码清单 5-11 中的程序。该程序对上一个程序进行了修改，实现了计算并显示最高分和最低分的功能。

代码清单 5-11

```
/*
 * 输入 5 名学生的分数并显示出其中的最高分和最低分
 */
#include <stdio.h>

#define NUMBER 5          /* 学生人数 */

int main(void)
{
    int i;
    int tensu[NUMBER];     /* 学生的分数 */
    int max, min;          /* 最高分和最低分 */

    puts("请输入学生的分数。");
    for (i = 0; i < NUMBER; i++) {
        printf("%2d 号 :", i + 1);
        scanf("%d", &tensu[i]);
    }

    min = max = tensu[0];
    for (i = 1; i < NUMBER; i++) {
        if(tensu[i] > max) max = tensu[i];
        if(tensu[i] < min) min = tensu[i];
    }

    printf("最高分 :%d\n", max);
    printf("最低分 :%d\n", min);

    return (0);
}
```

运行结果

请输入学生的分数。

1 号 : 95

2 号 : 83

3 号 : 85

4 号 : 63

5 号 : 89

最高分: 95

最低分: 63

请大家看一下右图所示部分。该语句的含义为将表达式 $max = tensu[0]$ 的判定结果赋给 min 。

表达式 $max = tensu[0]$ 的判定结果，就是赋值后 max 的值。 $tensu[0]$ 的值是 95，这个值会赋给 max ，而赋值表达式 $max = tensu[0]$ 的判定结果也就变成了 **int** 型的 95。该结果会赋给 min ，因此 min 和 max 值都变成了 $tensu[0]$ 的值，也就是 95。

C 语言中经常会使用这样的赋值方法。例如，使用 $a = b = 0$ ；就可以同时把 0 赋给 a 和 b 。

► 这仅仅是对赋值而言，对带有初始值的声明并不适用。不能像下面这样同时声明两个变量 a 和 b 。

```
int a = b = 0;
```

而需要像下面这样使用逗号分隔开声明。

```
int a = 0, b = 0;
```

或者也可以分两行进行声明。

```
int a = 0;
```

```
int b = 0;
```

取得最大值的步骤如下所示（最小值也是如此）。

```
max = tensu[0];
if (tensu[1] > max) max = tensu[1];
if (tensu[2] > max) max = tensu[2];
if (tensu[3] > max) max = tensu[3];
if (tensu[4] > max) max = tensu[4];
```

上述步骤与代码清单 3-12 中取得三个数的最大值的程序完全一样。只是对象整数从 3 个增加到了 5 个，从通过多个变量实现变成了通过一个数组来实现。

● 练习 5-6

假设变量 a 是 **double** 型，变量 b 是 **int** 型，请说明经过下述赋值后 a 和 b 的值分别是多少。

```
a = b = 1.5;
```

及格学生一览表

输入每个学生的分数，显示出 60 分以上学生的学籍编号和分数一览表。具体程序如代码清单 5-12 所示。这里不再对该程序进行说明，请大家自己理解。

代码清单 5-12

```
/*
    输入 5 名学生的分数并显示出 60 分以上学生的一览表
*/

#include <stdio.h>

#define NUMBER 5          /* 学生人数 */

int main(void)
{
    int i;
    int snum = 0;           /* 及格学生人数 */
    int tensu[NUMBER];     /* NUMBER 名学生的分数 */
    int succs[NUMBER];     /* 及格学生一览表（保存及格学生的下标）*/

    puts("请输入学生的分数。");
    for (i = 0; i < NUMBER; i++) {
        printf("%2d 号 :", i + 1);
        scanf("%d", &tensu[i]);
        if (tensu[i] >= 60)
            succs[snum++] = i; /* 添加到及格学生一览表中 */
    }

    puts ("及格学生一览表");
    puts ("-----");
    for (i = 0; i < snum; i++)
        printf("%2d 号 (%3d 分) \n", succs[i] + 1, tensu[succs[i]]);

    return (0);
}
```

运行结果

请输入学生的分数。

1 号 : 95

2 号 : 55

3 号 : 85

4 号 : 63

5 号 : 41

及格学生一览表

1 号 (95 分)

3 号 (85 分)

4 号 (63 分)

数组的元素个数

截至目前，我们看到的所有成绩处理程序中的学生人数都是 5。虽然通过定义宏来变更学生人数非常简单，但是每次都需要对程序进行修改，然后重新编译执行。

因此，我们可以定义一个比较大的数组，然后从头开始仅使用其中需要的部分。

后续代码清单 5-13 中的程序声明了一个包含 80 个元素的数组。程序执行的时候，可以输入 1 到 80 的人数作为变量 *num* 的值，仅使用数组最开头的 *num* 个元素。

成绩分布图

下述程序以 10 分为单位显示出了学生的成绩分布图。

代码清单 5-13

```

/*
  输入学生的分数并显示出分布情况
*/

#include <stdio.h>

#define NUMBER 80 /* 人数上限 */

int main(void)
{
    int i, j;
    int num; /* 实际的人数 */
    int tensu[NUMBER]; /* 学生的分数 */
    int bunpu[11] = {0}; /* 分布图 */

    printf("请输入学生人数:");
    do {
        scanf("%d", &num);
        if (num < 1 || num > NUMBER)
            printf("\a 人数范围 [1 到 %d]:", NUMBER);
    } while (num < 1 || num > NUMBER);

    puts("请输入学生的分数。");
    for (i = 0; i < num; i++) {
        printf("%2d 号:", i + 1);
        do {
            scanf("%d", &tensu[i]);
            if (tensu[i] < 0 || tensu[i] > 100)
                printf("\a 分数范围 [0 到 100]:");
        } while (tensu[i] < 0 || tensu[i] > 100);
        bunpu[tensu[i] / 10]++;
    }

    puts("\n □ 分布图 □");
    printf("      100:");
    for (j = 0; j < bunpu[10]; j++)
        putchar('*');
    putchar('\n');

    for (i = 9; i >= 0; i--) {
        printf("%3d - %3d:", i * 10, i * 10 + 9);
        for (j = 0; j < bunpu[i]; j++)
            putchar('*');
        putchar('\n');
    }

    return (0);
}

```

运行结果

```

请输入学生人数: 85 □
人数范围 [1 到 80]: 15 □
请输入学生的分数。

1 号: 17 □
2 号: 38 □
3 号: 100 □
4 号: 95 □
5 号: 23 □
6 号: 62 □
7 号: 77 □
8 号: 45 □
9 号: 69 □
10 号: 81 □
11 号: 83 □
12 号: 51 □
13 号: 42 □
14 号: 36 □
15 号: 60 □
□ 分布图 □
      100: *
      90-99: *
      80-89: **
      70-79: *
      60-69: ***
      50-59: *
      40-49: **
      30-39: **
      20-29: *
      10-19: *
       0-9:

```

5-2 多维数组

矩阵

代码清单 5-14 中的程序对下述 2 行 3 列矩阵 a 、 b 的和进行计算，并显示出了计算结果。

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad b = \begin{bmatrix} 6 & 3 & 4 \\ 5 & 1 & 2 \end{bmatrix}$$

代码清单 5-14

```
/*
 * 计算 2 行 3 列矩阵的和
 */
#include <stdio.h>

int main(void)
{
    int i, j;
    int ma[2][3] = { {1,2,3}, {4, 5, 6}};
    int mb[2][3] = { {6,3,4}, {5, 1, 2}};
    int mc[2][3] = { 0 };

    for (i = 0; i < 2; i++)
        for (j = 0; j < 3; j++)
            mc[i][j] = ma[i][j] + mb[i][j];

    for (i = 0; i < 2; i++) {
        for (j = 0; j < 3; j++)
            printf ("%3d", mc[i][j]);
        putchar ('\n');
    }

    return (0);
}
```

运行结果

```
7 5 7
9 6 8
```

本程序中使用的数组 ma 、 mb 和 mc 通常被称为二维数组。

二维数组和之前用到的数组（一维数组）的对比如图 5-4 所示。二维数组在概念上其实就是包含纵横二维的表单。

从图中我们可以了解到，二维数组中元素的

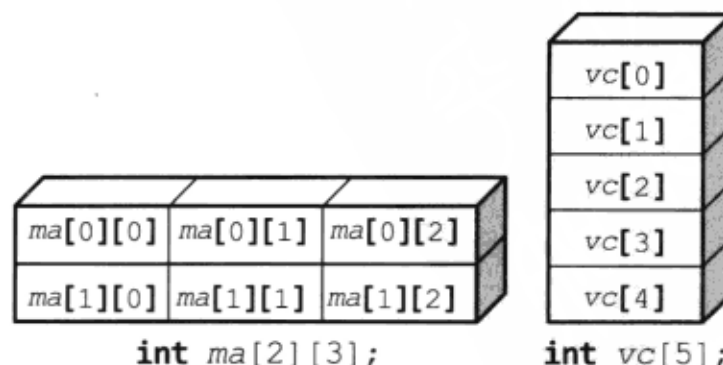


图 5-4 二维数组和一维数组

表现形式如下所示，必须要有两组下标。

数组名 [下标] [下标]

另外，维度在二以上的数组称为**多维数组**。

虽然概念上二维数组就是一个二维的表单，但是实际上数组中的元素是按照图 5-5 中的顺序连续排列的。

请大家看一下数组 *ma* 的声明语句，它在初始值的大括号中又嵌套了一层大括号结构。*ma*[0][0]、*ma*[0][1]、*ma*[0][2]、*ma*[1][0]、*ma*[1][1]、*ma*[1][2] 的初始值分别是 1、2、3、4、5、6（图 5-5 中的顺序）。

```
int ma[2][3] = { {1, 2, 3},
                {4, 5, 6},
                };
```

另外，该声明还可以写成右图的形式。

感觉上第二个逗号似乎没有什么必要，但是对于这样的书写格式，加上这个多余的逗号能够使程序看上去更加平衡一些。这样，初始值的结构也变得更加复杂了（图 5-6）。

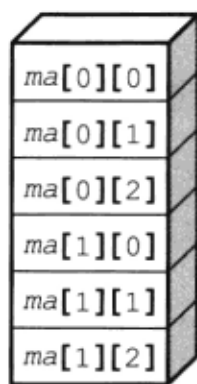


图 5-5 二维数组的元素排列

初始值

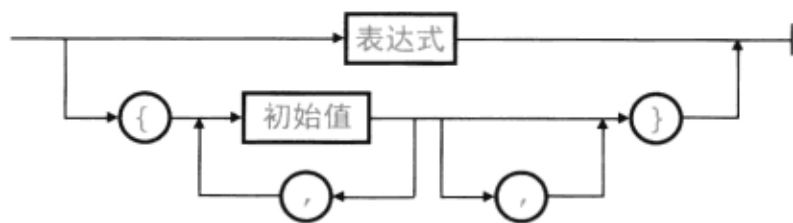


图 5-6 初始值的结构图

► 如果恰当地定义了初始值，声明数组的时候就可以省略最开始的元素个数。因此，数组 *ma* 也可以像下面这样进行声明。

```
int ma[][3] = {{1,2,3},{4,5,6}};
```

另外，初始值内部的大括号也可以省略。

```
int ma[2][3] = {1,2,3,4,5,6};
```

初始值不足时自动用 0 补足的规则也适用于多维数组，因此下面的声明

```
int mx[2][3] = {1,2,3,4,5};
```

等同于下述声明。当然，内嵌大括号中的初始值不足的时候也会进行同样的处理。

```
int mx[2][3] = {{1,2,3},{4,5,0}};
```

下面两条声明语句的效果是一样的。

```
int my[2][3] = {{1,2},{4}};
```

```
int my[2][3] = {{1,2,0},{4,0,0}};
```

● 练习 5-7

编写一段程序，求出矩阵 *x* 和 *y* 的积。

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$y = \begin{bmatrix} 1 & 5 \\ 5 & 3 \\ 8 & 1 \end{bmatrix}$$

5-3 质数计算

质数

质数是指在一个大于 1 的自然数中，除了 1 和它本身之外，无法被其他自然数整除的数。本节将会带领大家编写计算质数的程序。

质数计算程序（第 1 版）

判断一个数 no 是否为质数的方法就是看它能否被 2、3、……、 $no-1$ 整除。如果它不能被其中任何一个整数整除，那么这个数就是质数。例如我们要判断 13 是不是质数，就需要看它能否被下面这 11 个数整除。

2、3、4、5、6、7、8、9、10、11、12 /* 通过 11 次除法计算来判断其是否为质数 */

因为 13 无法被其中任何一个数整除，所以可以确定它是质数。

但是对于 12 来说，并不需要使用 2、3、4、5、6、7、8、9、10、11 这十个数来判断它是不是质数。也就是说，12 能够被第一个数 2 整除，所以它就不是质数。

~~2、3、4、5、6、7、8、9、10、11~~ /* 有一次被整除就可以判断出其不是质数 */

使用上述方法找出 1000 以内的质数。程序如代码清单 5-15 所示。

大整数

本程序中用来保存除法运算执行次数的变量 *counter* 被声明为 **unsigned long** 类型。它是 C 语言中表示范围最广的非负整数类型。输出此类型值时使用的转换说明是 "%lu"。

► **int** 类型的存储范围是 -32,767~32,767 (2-2 节)。

第 7 章将会具体学习关于 **unsigned long** 类型的知识。

break 语句

本程序的关键点就是 **break** 语句的动作。在程序流程中加入 **break** 语句，就可以跳出当前循环。

这样，当 no 除以 i 的时候（例如 12 除以 2 的时候），如果能整除（余数为 0），就中断

for 语句的循环，执行后续的 **if** 语句。这时，由于 *no* 和 *i* 并不相等，所以 *no* 不是质数。

► 如本程序所示，在内层循环中使用 **break** 语句的时候，将会跳出循环（但是并不能同时跳出外层循环）。

代码清单 5-15

运行结果

/*	
计算出 1000 以内的质数（第 1 版）	
*/	2
	3
#include <stdio.h>	5
	7
int main(void)	11
{	13
int i, no;	17
unsigned long counter = 0;	19
	23
for (no = 2; no <= 1000; no++) {	19
for (i = 2; i < no; i++) {	31
counter++;	37
if (no % i == 0) /* 能被整除的不是质数 */	41
break; /* 退出上述循环 */	43
}	(中略)
if (no == i) /* 直到最后也未被整除 */	983
printf("%d\n", no);	991
}	997
	乘除运算的次
printf(" 乘除运算的次数 : %lu\n", counter);	数: 78022
return (0);	
}	

► 由于后续程序中还会统计乘法的运算次数，因此显示出的信息中并没有使用“除法”而使用了“乘除”。

当 *no* 的值是 13 的时候，内层的 **for** 语句并没有中途结束，最后 *i* 的值变为 13。接下来执行后续的 **if** 语句，得出 *no* 是质数的判定结果，并把该数值显示出来。

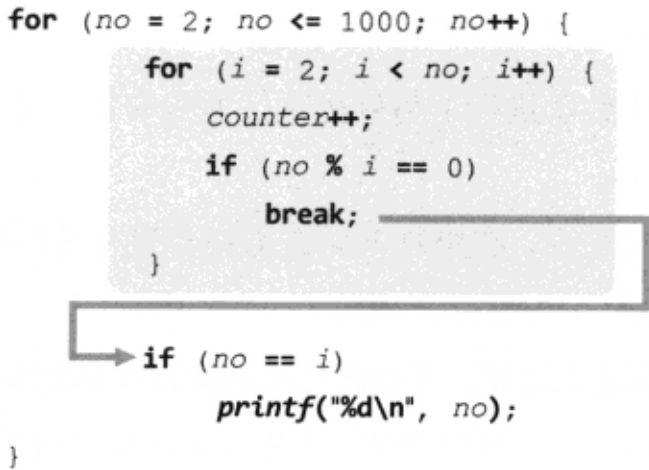


图 5-7 break 语句的动作

质数计算程序（第2版）

请大家注意下面的规律。

大于2的那些2的倍数（4、6、8、…）都不是质数。

既然我们已经知道了大于2的偶数都不是质数，就把它们排除在计算之外吧。程序如代码清单5-16所示。

代码清单 5-16

```
/*
    计算出1000以内的质数（第2版）
*/

#include <stdio.h>

int main(void)
{
    int i, no;
    unsigned long counter = 0;

    no = 2;
    printf("%d\n", no++);          /*2 是偶数中唯一的质数*/

    for ( ; no <= 1000; no += 2) { /* 只把奇数作为判断对象 */
        for (i = 2; i < no; i++) {
            counter++;
            if (no % i == 0)       /* 能被整除的不是质数 */
                break;           /* 退出上述循环 */
        }
        if (no == i)              /* 直到最后也未被整除 */
            printf("%d\n", no);
    }

    printf(" 乘除运算的次数 :%lu\n", counter);

    return (0);
}
```

运行结果

乘除运算的次数：77523

► 本章之后的程序只把运算次数作为运行结果显示出来。

本程序对偶数中唯一的质数2进行了特殊处理，并没有对它进行判断，而是直接显示出来了（这么做稍微有点投机取巧）。

对于程序的外层 **for** 语句来说，在循环开始之前没有进行任何操作，因此省略了表达式₁。

for（表达式₁；表达式₂；表达式₃）语句

本程序共进行了77 523次除法运算，并没有比第1版的78 022次减少多少，这有些出乎意料。

其实在第1版也曾进行过说明，例如判断12是否为质数的时候，在开始阶段就能通过被2整除判断出它不是质数，因此实际上第1版也只进行了一次除法运算。由于每次只省略了一次运算，所以除法运算的次数并没怎么减少。

质数计算程序（第3版）

结合之前的内容，我们来看一下下面的规律。

大于等于3的质数都无法被大于2的那些2的倍数（4、6、8、…）整除。

例如，判断13是否为质数的时候，不需要判断它能否被2、3、4、5、6、7、8、9、10、11、12这11个数整除。因为13是个奇数，所以无法被4或者8这样的数整除。因此，可以像下面这样只对奇数进行整除判断就可以了。具体程序如代码清单5-17所示。

2、3、4、5、6、7、8、9、10、11、12

代码清单 5-17

```

/*
    计算出1000以内的质数（第3版）
*/

#include <stdio.h>

int main(void)
{
    int i, no;
    unsigned long counter = 0;

    no = 2;
    printf("%d\n", no++);          /*2 是偶数中唯一的质数*/

    for ( ; no <= 1000; no += 2) {          /* 只把奇数作为判断对象 */
        for (i = 3; i < no; i+= 2) {        /* 只判断能否被奇数整除 */
            counter++;
            if (no % i == 0)                /* 能被整除的不是质数 */
                break;                      /* 退出上述循环 */
        }
        if (no == i)                        /* 直到最后也未被整除 */
            printf("%d\n", no);
    }

    printf(" 乘除运算的次数 :%lu\n", counter);

    return (0);
}

```

运行结果

乘除运算的次数: 38678

从字面上看，本程序仅仅对之前的版本的几个字符进行了修改。但是除法运算的次数却比第1版减少了将近一半，只有38678次。

虽然程序的外表也很重要，但更重要的是它的内容。

质数计算程序（第4版）

在第3版的基础上再进一步，就能够发现如下规律：

不能被3整除的整数也无法被大于3的那些3的倍数（6、9、……）整除。
 不能被5整除的整数也无法被大于5的那些5的倍数（10、15、……）整除。
 ⋮

这样，判断整数 *no* 是否为质数的时候，就不需要去判断它是否能被质数之外的数字整除了。如果它能被6或者10整除，那应该也可以被它们的约数3或者5整除。

因此，如果整数 *no* 满足下述条件，那它就是质数。

无法被小于 *no* 的质数整除。

实现该算法的程序如代码清单5-18所示。

首先通过两个赋值语句把2和3赋给数组 *prime* 的开头两个元素，同时 *ptr* 的值增加为2[图5-8a，图5-8b]。

因为数组 *prime* 是用来保存已知质数的，所以 *ptr* 记录了质数的个数。

另外，我们知道4并不是质数，因此只需要判断5到999中的奇数是不是质数就可以了（通过外层的 **for** 语句来控制）。

内层的 **for** 语句用来判断 *no* 是否为质数，具体来说就是用来判断 *no* 是否能被已知的质数整除。不过，由于 *no* 肯定是奇数，无法被2整除，所以可以直接判断它能否被3以上的质数整除。

当 *no* 的值分别为5、7、……、15时，处理流程如下所示（*no* 值为9和15时都可以被3整除）。

<i>no</i> 为5的时候	3	/* 通过1次除法运算判断出它是质数 */
<i>no</i> 为7的时候	3、5	/* 通过2次除法运算判断出它是质数 */
<i>no</i> 为9的时候	3、5、7	/* 通过1次除法运算判断出它不是质数 */
<i>no</i> 为11的时候	3、5、7	/* 通过3次除法运算判断出它是质数 */
<i>no</i> 为13的时候	3、5、7、11	/* 通过4次除法运算判断出它是质数 */
<i>no</i> 为15的时候	3、5、7、11、13	/* 通过1次除法运算判断出它不是质数 */

当然，3、5、……这些除数都保存在数组 *prime*[1]、*prime*[2]、……*prime*[*ptr*-1] 中（前面提到过没有必要把 *prime*[0] 的值2作为除数进行计算）。内层 **for** 语句用来进行上述控制。另外，如果没能被任何除数整除，则通过下面语句把得到的新质数 *no* 添加到数组 *prime* 中。

```
if (ptr == i)
    prime[ptr++] = no;
```

代码清单 5-18

```

/*
    计算出 1000 以内的质数 (第 4 版)
*/

#include <stdio.h>

int main(void)
{
    int i, no;
    int prime[500];
    int ptr = 0;
    unsigned long counter = 0;

    prime[ptr++] = 2;
    prime[ptr++] = 3;

    for (no = 5; no <= 1000; no += 2) {
        for (i = 1; i < ptr; i++) {
            counter++;
            if (no % prime[i] == 0)
                break;
        }
        if (ptr == i)
            prime[ptr++] = no;
    }

    for (i = 0; i < ptr; i++)
        printf("%d\n", prime[i]);

    printf("  乘除运算的次数: %lu\n", counter);

    return (0);
}

```

运行结果

乘除运算的次数: 14622

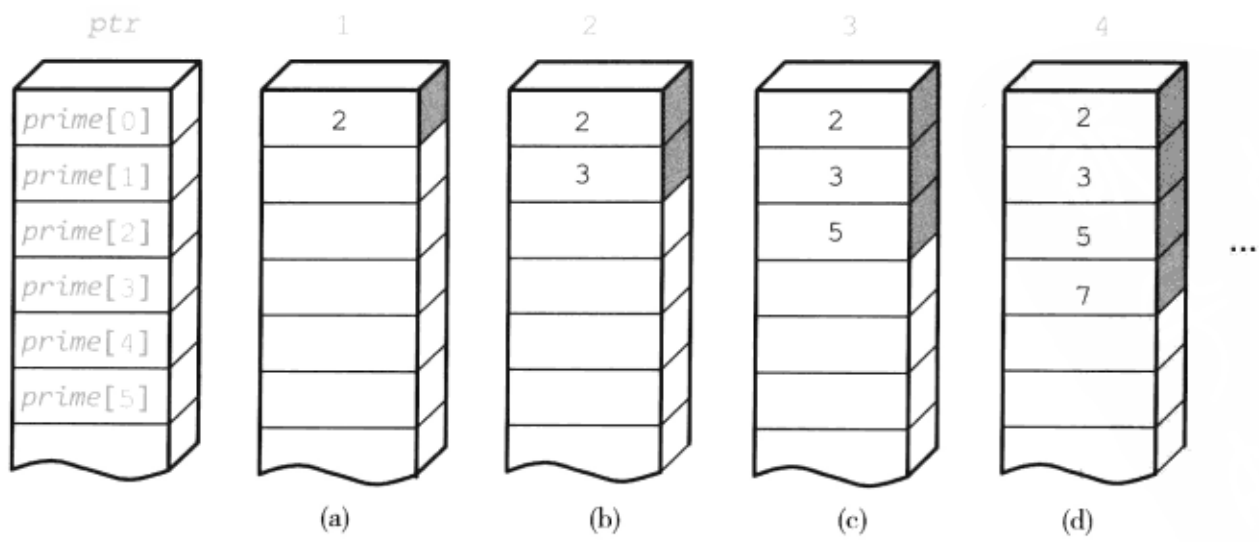


图 5-8 用于存储质数的数组的变化

质数计算程序（第5版）

100 的约数排列如下：

2×50
 4×25
 5×20
 10×10
 20×5
 25×4
 50×2

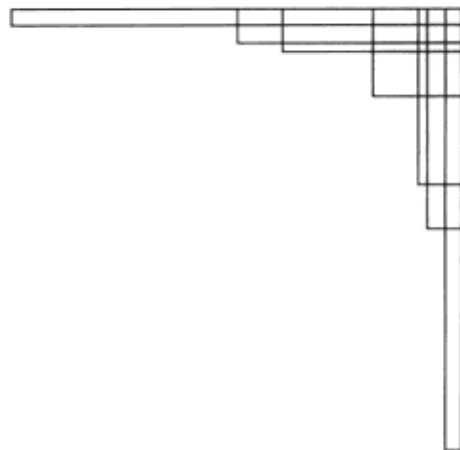


图 5-9 长方形和约数

如果把 100 作为长方形的面积，那么这些值就是它的长度和宽度。我们可以看到上述排列以正方形 10×10 为中心形成对称（图 5-9）。例如， 5×20 和 20×5 其实是同一个长方形，只不过长和宽颠倒了一下而已。因此，如果 100 不能被 5 整除，那么它也无法被 20 整除。

我们可以对截至正方形边长（这里是 10）的数值进行除法运算，通过能否被整除来判断该数字是否为质数。也就是说，可以根据以下标准来判断 no 是不是质数。

无法被小于等于 no 的平方根的质数整除。

实现上述算法的程序如代码清单 5-19 所示。

逗号运算符

本程序的关键部分就是 **for** 语句的控制表达式。

```
counter++, prime[i] * prime[i] <= no
```

其中的逗号被称为**逗号运算符**（comma operator），如表 5-1 所示。通常，使用逗号运算符的表达式 $op1$ ， $op2$ 会顺次对表达式 $op1$ 和 $op2$ 进行计算，并把 $op2$ 的类型和值作为整个表达式的类型和结果。虽然也会对逗号左边的表达式进行计算，但是计算结果会被舍弃。例如当 i 的值为 3， j 的值为 5 的时候，表达式 $i++$ ， j 的计算结果就是第二个操作数的值 5。同时， i 的值也会正常地自动增加，变为 4。

■ 表 5-1 逗号运算符

逗号运算符	a, b	顺次对 a 和 b 进行计算，并把 b 的计算结果作为整个表达式的值。
-------	--------	---

代码清单 5-19

```

/*
    计算出 1000 以内的质数 (第 5 版)
*/

#include <stdio.h>

int main(void)
{
    int i, no;
    int prime[500];
    int ptr = 0;
    unsigned long counter = 0;

    prime[ptr++] = 2;
    prime[ptr++] = 3;

    for (no = 5; no <= 1000; no += 2) {
        int flag = 0;
        for (i = 1; counter++, prime[i] * prime[i] <= no; i++) {
            counter++;
            if (no % prime[i] == 0) {
                flag = 1;
                break;
            }
        }
        if (!flag)
            prime[ptr++] = no;
    }

    for (i = 0; i < ptr; i++)
        printf("%d\n", prime[i]);

    printf("  乘除运算的次数: %lu\n", counter);

    return (0);
}

```

运行结果

乘除运算的次数: 3774

在执行上述程序的乘法运算之前, 首先 *counter* 的值会自动增加。由于该值会被舍弃, 因此是否继续执行 **for** 语句的循环, 还是要通过逗号运算符的第二个操作数 *prime[i] * prime[i] <= no* 是否成立来判断。

► 关于逗号运算符, 大家还可以参考 8-1 节。



第 6 章

函 数

在前几章的程序中，我们通过 **printf**、**puts**、**putchar** 函数实现了输出显示的功能，通过 **scanf** 函数实现了读取键盘输入信息的功能。也就是说，在进行显示等输入输出处理的时候，我们都对函数发出了“之后就拜托你了”这样的请求。但是，像这样只“依靠他人”是无法编写出完整的程序的。

本章将带领大家学习函数的相关知识。



6-1 什么是函数

main 函数

截至目前，大家见过的程序格式都如图 6-1 所示。其中蓝色底纹的部分是固定代码，只有除此之外的内容才是由编程人员自己编写的。

图中蓝色字体的部分称为 **main** 函数 (main function)。原则上每个 C 语言程序中必须并且只能存在一个 **main** 函数。

程序运行的时候，会执行 **main** 函数的主体部分。因此，C 语言编程都是从编写 **main** 函数开始的。

```
#include <stdio.h>

int main(void)
{
    /*... 中略 ...*/
    return (0);
}
```

图 6-1 固定代码和 main 函数

库函数

目前为止我们已经使用过 **printf**、**scanf**、**puts**、**putchar** 等函数。由 C 语言提供的这些函数称为库函数 (library function)。

► 通常各个编译器在提供 C 语言规定的函数之外，还会提供各自不同的函数。具体内容请参考各编译器的说明书。

函数定义和函数调用

当然，我们也可以自己来创建函数。或者说，实际上我们必须亲自动手创建各种函数。那么首先来尝试一下比较简单的函数。

创建一个函数，接收两个整数参数，返回较大整数的值。

程序如代码清单 6-1 所示。

本程序由 *maxof* 函数和 **main** 函数组成。函数 *maxof* 的函数定义 (function definition) 结构如图 6-2 所示。

首先，我们来看一下函数头 (function header) 的部分。返回类型是 **int**，表示本函数会返回 **int** 型的值。后面紧跟着的 *maxof* 是函数的名称。

小括号中的部分是函数的形式参数 (形参, parameter) 的声明。这样，就定义出了两个

int 型的形参 *x* 和 *y*。

有多个形参的时候可以使用逗号隔开。

代码清单 6-1

```
/*
 * 返回两个整数中较大值的函数
 */

#include <stdio.h>

/* 返回较大整数的值 --- */
int maxof(int x, int y)
{
    if (x > y)
        return (x);
    else
        return (y);
}

int main(void)
{
    int na, nb;

    puts("请输入两个整数。");
    printf("整数 1:");    scanf("%d", &na);
    printf("整数 2:");    scanf("%d", &nb);

    printf("较大整数的值是 %d。\\n", maxof(na, nb));

    return (0);
}
```

运行结果

请输入两个整数。
整数 1: 83
整数 2: 45
较大整数的值是 83。

使用函数，其实就是一个调用该函数的过程。调用函数的表达式称为函数调用表达式 (function call expression)。

请大家看一下图 6-3 中的函数调用表达式，它发出了下面这样的请求：

函数 *maxof*，这里给你传递了 *na* 和 *nb* 两个值，请返回其中较大的值。

对于函数 *maxof* 来说，它会根据传递来的两个 **int** 型实参，返回一个 **int** 型的值。

另外，用来包围实参的小括号 () 称为函数调用运算符 (function call operator)。

	返回类型	函数名	形参声明
函数头	int	maxof	(int x, int y)
	{		
		if (x > y)	
		return (x);	
		else	
函数体		return (y);	
	}		

图 6-2 函数定义结构

函数名	实参	实参
maxof	(na , nb)	
函数调用运算符		

图 6-3 函数调用表达式

假设 na 的值为 24, nb 的值为 62, 调用函数时参数传递的过程如图 6-4 所示。进行函数调用的时候, 程序流会从调用函数一侧转移到被调用的函数一侧。这时, 实参的值会被复制到形参中。因此, 在调用 `maxof` 函数, 程序流转入到函数体中的时候, 会把 24 和 62 这两个值赋给形参 x 和 y 。

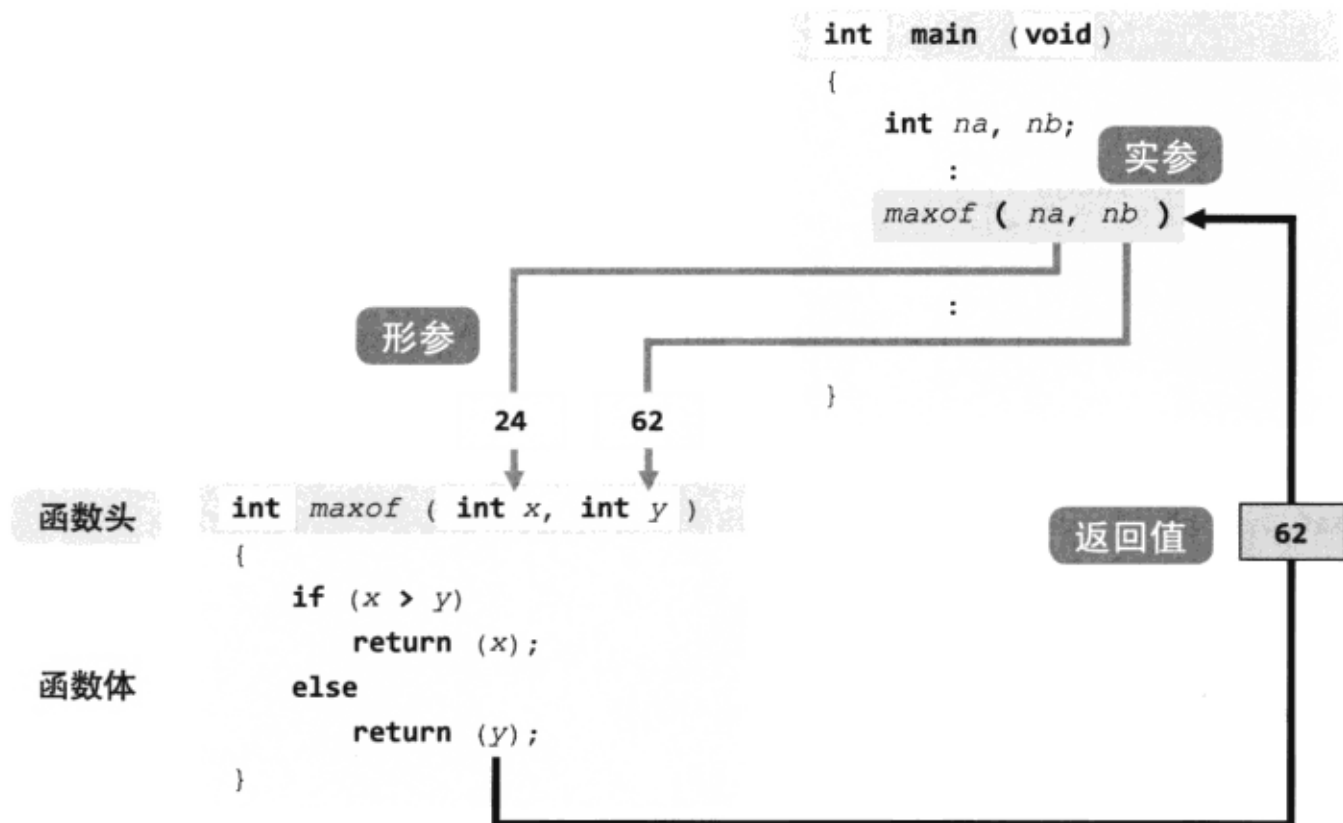


图 6-4 函数调用和参数传递

函数体 (function body) 一定是被大括号 `{}` 括起来的复合语句。不论是 `maxof` 函数, 还是 `main` 函数, 都符合这样的要求。

对于函数体来说, 如果 x 大于 y , 则关系表达式 $x > y$ 的判断结果为 1, 执行 `return (x);` 语句。否则就执行 `return (y);` 语句。这样的语句称为 **return 语句** (return statement) 它的语法结构如图 6-5 所示。

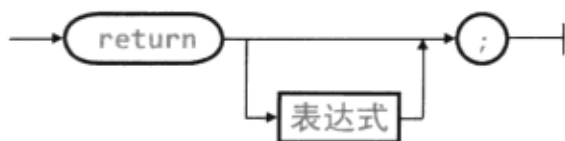


图 6-5 return 语句的结构图

► 如果省略 `return` 后面的表达式写作 `return;`, 则会返回一个未定义的值。另外, 从结构图中我们可以看出, 表达式并不一定要使用小括号括起来。但为了使前后关系更加清晰, 本书中使用了小括号对表达式进行标识。

程序流在遇到 **return** 语句，或者执行到函数体最后的大括号时，会从该函数跳转到调用函数。同时，书写在 **return** 后面的表达式的值会被返回（可以想象成带了个小礼物回来）。如果 *na* 和 *nb* 的值分别为 24 和 62，则函数 *maxof* 会返回 *y* 的值，也就是 62。

另外，请大家牢记下述内容。

■ 注意 ■

对函数调用表达式进行判定的时候，会得到该返回类型的返回值。

因此，如果 *na* 和 *nb* 的值分别为 24 和 62，则函数调用表达式 *maxof(na, nb)* 的判定结果为 **int** 型的 62。

► 函数调用时传递的只是参数的值，因此调用函数时使用的实参既可以是变量，也可以是常量。例如，对于下述语句来说，会把 *x* 和 5 当中较大的值赋给 *c*。

```
c = maxof(x, 5);
```

另外，大家可以完全不用担心实参和形参的名字相同的问题。

即使像 *maxof* 这样非常短小的函数，也可以有多种定义方式。下面这些函数，都能实现同样的功能。当然，其中最简洁的还是使用条件运算符（3-1 节）的 (c)。

<p>(a)</p> <pre>int maxof(int x, int y) { int max; if (x > y) max = x; else max = y; return (max); }</pre>	<p>(b)</p> <pre>int maxof(int x, int y) { int max = x; if (y > max) max = y; return (max); }</pre> <p>(c)</p> <pre>int maxof(int x, int y) { return ((x > y) ? x : y); }</pre>
--	---

(a) 和 (b) 中的程序，为了保存较大的值，都使用了变量 *max*。像这样只在某个函数中使用的变量，原则上需要在该函数中进行声明。但是，该变量不能与形参同名，否则会发生变量名冲突的错误。

上述函数与代码清单 6-1 中的不同，只有一个 **return** 语句。

函数的入口只有一个。因此，如果有多个出口的话，阅读起来就会比较困难，还是统一起来更好一些。

三个数中的最大值

显示三个整数中最大值的程序如代码清单 6-2 所示。

代码清单 6-2

```
/*
 * 返回三个整数中最大值的函数
 */

#include <stdio.h>

/*--- 返回三个整数中的最大值 ---*/
int max3(int x, int y, int z)
{
    int max = x;

    if (y > max) max = y;
    if (z > max) max = z;
    return (max);
}

int main(void)
{
    int na, nb, nc;

    puts("请输入三个整数。");
    printf("整数 1:");    scanf("%d", &na);
    printf("整数 2:");    scanf("%d", &nb);
    printf("整数 3:");    scanf("%d", &nc);

    printf("最大值是 %d。\\n", max3(na, nb, nc));

    return (0);
}
```

运行结果

请输入三个整数。

整数 1: 83

整数 2: 45

整数 3: 25

最大值是 83。

这里就不用再做过多说明了。

● 练习 6-1

创建一个函数，返回两个 **int** 型整数中较小一数的值。

```
int minof (int x, int y) { /* ... */ }
```

为了确认函数的动作，还需要大家创建一个合适的 **main** 函数来组成一段完整的程序（之后的练习也是如此）。

● 练习 6-2

创建一个函数，返回三个 **int** 型整数中的最小值。

```
int min3 (int x, int y, int z) { /* ... */ }
```


平方差

输入两个整数，计算它们的平方差并显示。程序如代码清单 6-3 所示。

代码清单 6-3

```
/*
    计算两个整数的平方差
*/

#include <stdio.h>

/*--- 返回平方 ---*/
int sqr(int x)
{
    return (x * x);
}

/*--- 返回差值 ---*/
int diff(int x, int y)
{
    return (x > y ? x - y : y - x);
}

int main(void)
{
    int kx, ky, kx2, ky2;

    puts("请输入两个整数。");
    printf("整数 kx:"); scanf("%d", &kx);
    printf("整数 ky:"); scanf("%d", &ky);

    kx2 = sqr(kx); /*kx 的平方*/
    ky2 = sqr(ky); /*ky 的平方*/
    printf("kx 和 ky 的平方差是 %d.\n", diff(kx2, ky2));

    return (0);
}
```

运行结果

请输入两个整数。

整数 kx: 5

整数 ky: 4

kx 和 ky 的平方差是 9。

6

本程序将 kx 的平方和 ky 的平方临时存储在变量 $kx2$ 和 $ky2$ 中。但是，由于这些变量之后都不会再使用了，因此也可以像下面这样调用函数 `diff`。

```
diff (sqr(kx), sqr(ky))
```

直接把函数 `sqr` 的返回值传递给函数 `diff`。

► 接下来把函数 `diff` 的返回值原样传递给 `printf` 函数。

● 练习 6-3

创建一个函数，返回 `int` 型整数的立方。

```
int cube(int x) { /* ... */ }
```

幂

下面我们来创建一个计算幂的函数。如果 no 是整数，则通过对 dx 进行 no 次乘法运算得出 dx 的 no 次方。程序如代码清单 6-4 所示。

► 例如，4.0 的三次幂就是 $4.0 \times 4.0 \times 4.0 = 64.0$ 。

代码清单 6-4

```
/*
  计算幂
*/

#include <stdio.h>

/*--- 返回 dx 的 no 次幂 ---*/
double power(double dx, int no)
{
    int i;
    double tmp = 1.0;

    for (i = 1; i <= no; i++)
        tmp *= dx;
    return (tmp);
}

int main(void)
{
    int n;
    double x;

    printf("请输入一个实数:"); scanf("%lf", &x);
    printf("请输入一个整数:"); scanf("%d", &n);

    printf("%.2f 的 %d 次幂是 %.2f。 \n", x, n, power(x, n));

    return (0);
}
```

运行结果

请输入一个实数: 4.0

请输入一个整数: 3

4.00 的 3 次幂是 64.00。

值传递

像前面程序那样通过值来进行参数传递的机制称为值传递 (pass by value)。因此，即使在被调用函数内对接收到的形参的值进行修改，也不会对调用函数的实参产生任何影响。

这就好比把本页的内容复印出来，不论用红笔在上面写些什么，都不会对原书产生任何影响一样。在函数 `power` 中，可以对 dx 或者 no 的值进行任意修改。

对 dx 的值进行 no 次乘法运算， no 的值就会像 5、4、……、1 这样逐渐递减。

这样修改后的函数 `power` 如代码清单 6-5 所示。

代码清单 6-5

```

/*--- 返回 dx 的 no 次幂 (第 2 版) ---*/
double power(double dx, int no)
{
    double tmp = 1.0;

    while (no-- > 0)
        tmp *= dx;
    return (tmp);
}

```

► 这里没有给出 **main** 函数, 请大家模仿代码清单 6-4 补全。

像这样不使用循环变量 *i*, 可以使得函数更加简洁紧凑。

■ 注意 ■

灵活运用值传递的优点, 可以让函数更加简洁紧凑。

调用其他函数

截止到目前的程序, 都是在 **main** 函数中调用库函数或者我们自己创建的函数。当然, 在自己创建的函数中也可以调用其他函数。

使用代码清单 6-1 中的函数 *maxof*, 返回四个整数最大值的函数如代码清单 6-6 所示。

代码清单 6-6

```

/*--- 返回四个整数的最大值 ---*/
int max4(int w, int x, int y, int z)
{
    return (maxof(maxof(w, x), maxof(y, z)));
}

```

该函数对 *w*、*x* 中的较大值和 *y*、*z* 中的较大值进行比较, 返回其中较大的值。

我们可以认为函数就是程序的一个零件 (例如, 想要实现显示功能的时候, 就调用 **printf** 这个零件)。在制作零件的时候, 如果有其他方便的零件, 我们也可以大量地使用。

● 练习 6-4

使用代码清单 6-3 中的 *sqr* 函数创建另一个函数, 返回 **int** 型整数的四次幂。

```
int pow4 (int x) { /* ... */ }
```

6-2 函数设计

没有返回值的函数

在第4章，我们编写了一段可以通过排列*显示出直角三角形的程序。下面我们把连续显示任意个数*的部分单独做成一个函数，并通过调用它来显示出一个直角在左下方的直角三角形。完成后的程序如代码清单6-7所示。

代码清单6-7

```
/*
    显示出一个直角在左下方的直角三角形（函数版）
*/
#include <stdio.h>

/*---- * 连续显示出 no 个 ----*/
void put_stars(int no)
{
    while (no--> 0)
        putchar('*');
}

int main(void)
{
    int i, ln;

    printf("三角形有几层: ");
    scanf("%d", &ln);

    for (i = 1; i <= ln; i++) {
        put_stars(i);
        putchar('\n');
    }

    return (0);
}
```

运行结果

```
三角形有几层: 5
*
**
***
****
*****
```

/* --- 参考: 代码清单4-18 --- */
for (i = 1; i <= ln; i++) {
 for (j = 1; j <= i; j++)
 putchar('*');
 putchar('\n');
}

本函数只是用来进行显示的，因此没有需要返回的结果。这种没有返回值的函数类型，要声明为 **void**。

► **void** 就是“空”的意思。在 C 语言中，不论有没有返回值都同样称为函数。而在其他编程语言中，没有返回值的会定义为其他非函数的概念，例如子程序（Fortran）或者过程（Pascal）。

通用性

通过使用函数 `put_stars` 可以把实现三角形显示的二重循环简化为一重循环，从而提高

程序的可读性。

显示直角在右下角的直角三角形的程序如代码清单 6-8 所示。

代码清单 6-8

```
/*
  显示直角在右下角的直角三角形（函数版）
*/
#include <stdio.h>

/*--- 连续显示 no 个字符 ch---*/
void put_nchar(int ch, int no)
{
    while (no-- > 0)
        putchar(ch);
}

int main(void)
{
    int i, ln;

    printf("三角形有几层 :");
    scanf("%d", &ln);

    for (i = 1; i <= ln; i++) {
        put_nchar(' ', ln - i);
        put_nchar('*', i);
        putchar('\n');
    }

    return (0);
}
```

运行结果

三角形有几层: 5

```

      *
     **
    ***
   ****
  *****

```

/*--- 参考: 代码清单 4-19 ---*/

```

for (i = 1; i <= ln; i++) {
    for (j = 1; j <= ln-i; j++)
        putchar(' ');
    for (j = 1; j <= i; j++)
        putchar('*');
    putchar('\n');
}

```

本程序还需要连续显示空白字符，因此需要创建另一个函数 `put_nchar` 来代替函数 `put_stars`。该函数可以连续显示出 `no` 个通过形参传递来的字符。

► 之前给大家介绍过字符常量是 `int` 型的（4-2 节）。除此之外，还存在显示字符的 `char` 型。关于 `char` 型我们将会在第 7 章进行说明。

函数 `put_nchar` 和只能显示 `*` 的函数 `put_stars` 比起来，具有更加通用的优势。

当然，如果有必要的话，我们也可以像右面这样定义函数 `put_stars`（不用说，还是需要使用函数 `put_nchar`）。

```
/*--- 连续显示 no 个 ---*/
void put_stars(int no)
{
    put_nchar('*', no);
}
```

● 练习 6-5

创建一个函数，连续发出 `no` 次警报。

```
void alert (int no) { /* ... */ }
```

不含形参的函数

输入一个非负整数并显示其倒转之后的值。程序如代码清单 6-9 所示。该程序是由代码清单 4-2 中的程序修改而来的。

代码清单 6-9

```
/*
    逆向显示输入的非负整数
*/

#include <stdio.h>

/*--- 返回输入的非负整数 ---*/
int scan_uint(void)
{
    int tmp;

    do {
        printf("请输入一个非负整数: ");
        scanf("%d", &tmp);
        if (tmp < 0)
            puts("\a 请不要输入负整数。");
    } while (tmp < 0);
    return (tmp);
}

/*--- 返回非负整数倒转后的值 ---*/
int rev_int(int num)
{
    int tmp = 0;

    if (num > 0) {
        do {
            tmp = tmp * 10 + num % 10;
            num /= 10;
        } while (num > 0);
    }
    return (tmp);
}

int main(void)
{
    int nx = scan_uint();

    printf("该整数倒转后的值是 %d。\\n", rev_int(nx));

    return (0);
}
```

运行结果

请输入一个非负整数: 5

请不要输入负整数。

请输入一个非负整数: 118

该整数倒转后的值是 811。

函数 `scan_uint` 读取从键盘输入的非负整数并返回。该函数不接收形参，为了加以说明，在小括号中写入了 `void`。

- ▶ 固定程序 `int main(void)` 表示 `main` 函数不包含形参（另外也存在包含形参的 `main` 函数）。

函数返回值的初始化

请大家注意 `main` 函数中声明变量 `nx` 的部分。该变量的初始值是函数 `scan_uint()` 的调用表达式。

像这样将函数的返回值作为变量初始值也是可以的。

- ▶ 但是，只有拥有自动存储期的对象（将在 6-3 节为大家介绍）可以这样进行初始化。

作用域

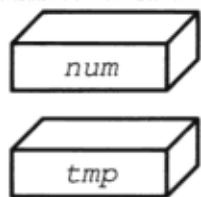
函数 `scan_uint` 和函数 `rev_int` 都包含一个拥有相同标识符（名称）的变量 `tmp`，但它们却是各自独立的不同变量（图 6-6）。

也就是说，函数 `scan_uint` 中的变量 `tmp` 是函数 `scan_uint` 特有的变量。而函数 `rev_int` 中的变量 `tmp` 是函数 `rev_int` 中特有的变量。

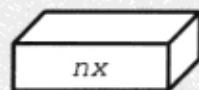
赋给变量的标识符，它的名称都有一个通用的范围，称为**作用域**（scope）。

在程序块（复合语句）中声明的变量的名称，只在该程序块中通用，在其他区域都无效。也就是说，从变量声明的位置开始，到包含该声明的程序块最后的大括号 `}` 为止这一区间内通用。这样的作用域称为**块作用域**（block scope）。

```
int rev_int(int num)
{
    int tmp = 0;
    /* ... */
    return (tmp);
}
```



```
int main(void)
{
    int nx = scan_uint();
    /* ... */
}
```



```
int scan_uint(void)
{
    int tmp;
    /* ... */
    return (tmp);
}
```

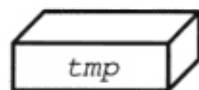


图 6-6 在函数内声明的变量

● 练习 6-6

创建一个函数，在屏幕上显示出“你好。”并换行。

```
void hello(void) { /* ... */ }
```

计算最高分的程序

输入 5 名学生的分数，显示出其中最高分。程序如代码清单 6-10 所示。

代码清单 6-10

```
/*
    计算最高分
*/

#include <stdio.h>

#define NUMBER    5

int  tensu[NUMBER];

int  top(void);          /* [ 声明函数类型 ] */

int main(void)
{
    extern int  tensu[];
    int  i;

    printf(" 请输入 %d 名学生的分数。\\n", NUMBER);
    for (i = 0; i < NUMBER; i++) {
        printf("%d:", i + 1);
        scanf("%d", &tensu[i]);
    }
    printf(" 最高分 = %d\\n", top());
    return (0);
}

/* --- 返回数组 tensu 的最大值 (函数定义) --- */
int top(void)
{
    extern int  tensu[];
    int  i;
    int  max = tensu[0];
    for (i = 1; i < NUMBER; i++)
        if (tensu[i] > max)
            max = tensu[i];
    return (max);
}
```

运行结果

请输入 5 名学生的分数。

1 : 53

2 : 49

3 : 21

4 : 91

5 : 77

最高分 = 91

文件作用域

在函数的程序块中声明的变量等标识符是该程序块特有的部分。而像数组 *tensu* 这样，在函数外声明的变量标识符，其名称从声明的位置开始，到该程序的结尾都是通用的。

这样的作用域称为文件作用域（file scope）。

声明和定义

程序中 `int tensu [NUMBER];` 的声明，创建了一个元素数为5，元素类型为 `int` 的数组。像这样创建变量实体的声明称为定义（definition）声明。另外，使用了 `extern` 的蓝色底纹部分的声明表示“使用的是在某处创建的 `tensu`”。这里并没有真正创建出变量的实体，因此称为非定义声明。

由于数组 `tensu` 是在函数外定义的，所以只需要在 `main` 函数或函数 `top` 中明确声明要使用它，就可以放心地用了。

► 由于数组 `tensu` 被赋予了文件作用域，所以在 `main` 函数和函数 `top` 中无需特意声明，可以直接使用。因此，程序中蓝色底纹部分可以省略。

函数原型声明

本程序中函数 `top` 的函数定义在 `main` 函数之后，这就需要我们提前通知 `main` 函数，函数 `top` 无需参数，并且会返回 `int` 型的值。因此，需要使用 `int top(void);` 的声明。

像这样明确记述了函数的返回类型，以及形参的类型和个数等的声明称为函数原型声明（function prototype declaration）。

► 需要注意的是该声明要以分号结尾。

函数原型声明只声明了函数的返回值和形参等相关信息，并没有定义函数的实体。用来定义实体的是函数定义。

通过本例大家应该可以明白定义声明和非定义声明的不同之处了吧。

另外，如果函数 `top` 的需求（返回值的类型和形式参数等）发生了改变，那么函数定义和函数原型声明两部分都必须进行修改。

但是，在编写程序的时候，如果把函数 `top` 的函数定义放在 `main` 函数之前，就不用特意使用函数原型声明了。

一般，把 `main` 函数放在程序最后的位置，而把将被调用的函数放在程序前部是比较好的选择。

■ 推荐 ■

把被调用的函数放在调用函数之前。

头文件和文件包含指令

通过函数原型声明，可以指定函数的参数以及返回值的类型等信息，这样就可以放心地调用该函数了。

库函数 **printf** 或者 **puts** 等的函数原型声明都包含在 `<stdio.h>` 中，因此必须要使用下述固定的指令。

```
#include <stdio.h>
```

通过 **#include** 指令，就可以把 `<stdio.h>` 中的全部内容都读取到程序中（图 6-7）。包含库函数的函数原型声明的 `<stdio.h>` 称为**头文件**（header），而取得头文件内容的 **#include** 指令称为**文件包含指令**。

- 不同的编译器实现头文件的方法也有所不同（也不能保证会提供单独的文件来提供）。

#include 指令行引入该头文件的内容

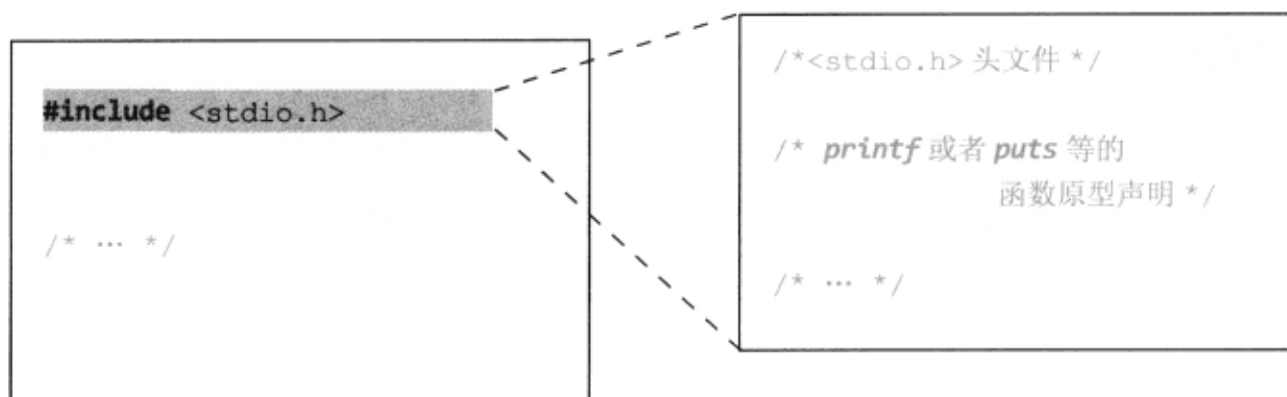


图 6-7 头文件的文件包含指令

例如，**putchar** 函数的函数原型声明在头文件 `<stdio.h>` 中的声明格式如下所示：

```
int putchar(int __c);
```

- 编译器不同，形参的名称也有可能不同。另外，由于可以在函数原型声明的时候不指定形参的名称，所以下面这样的声明也是可以的。

```
int putchar(int);
```

同样，本章一开始时提到的函数 *maxof* 的函数原型声明也可以写成下面这样：

```
int maxof(int, int);
```

函数的通用性

函数 *top* 的工作过程如下所示：

找出 **int** 型数组 *tensu* 最前面 **NUMBER** 个元素中的最大值，然后返回该值。

不了解程序内容的人看到上述说明的时候，可能会问“数组 *tensu* 是什么”、“**NUMBER** 是多少呢”。确实，只有编写程序的人才明白这些名称的含义。

本程序只对单一科目的分数进行计算，可如果想要计算英语、数学等各个科目的最高分的话，又该怎么办呢？

另外，如果英语是选修科目，数学是必修科目，当每科的人数都不相同的时候，又该如何处理呢？

对于上述要求，函数 *top* 都无法满足。所以说从函数的通用性考虑，至少应该满足下面两个条件：

■ 可以处理任意数组

不仅可以处理数组 *tensu*，而且也可以处理其他任意数组。

■ 可以处理不同元素个数的数组

数组的元素个数不仅仅只有 **NUMBER**（即 5 个），这就还要可以指定数组的元素个数（也就是学生人数）。

满足上述条件的程序如代码清单 6-11 所示。

► 后面会对程序进行详细说明。另外，数学和英语的学生人数都是 **NUMBER**，也就是 5。但是，这里使用的函数 *max_of* 本身和学生人数并没有依存关系。

函数 *max_of* 动作如下：

找出包含任意个元素的 **int** 型数组中元素的最大值，然后返回该值。

像函数 *top* 一样，无需使用 *tensu* 或者 **NUMBER** 等特定的名称就可以进行说明了。请大家注意下述事项。

■ 注意 ■

进行函数设计的时候，应该尽量提高其通用性。

这样一来，就可以更加简洁地说明函数功能。

数组的传递

在代码清单 6-11 的程序中, 使用数组 *eng* 来存储英语的分数, 使用数组 *mat* 来存储数学的分数。它们的最高分分别保存在变量 *max_e* 和 *max_m* 中。

代码清单 6-11

```
/*
    计算英语分数和数学分数中的最高分
*/

#include <stdio.h>

#define NUMBER 5

/*--- 返回元素个数为 no 的数组 vc 中的最大值 ---*/

int max_of(int vc[], int no)
{
    int i;
    int max = vc[0];
    for (i = 1; i < no; i++)
        if (vc[i] > max)
            max = vc[i];
    return (max);
}

int main(void)
{
    int i;
    int eng[NUMBER];          /* 英语的分数 */
    int mat[NUMBER];          /* 数学的分数 */
    int max_e, max_m;         /* 最高分 */

    printf("请输入 %d 名学生的分数。\\n", NUMBER);
    for (i = 0; i < NUMBER; i++) {
        printf("[%d] 英语 :", i + 1);      scanf("%d", &eng[i]);
        printf(" 数学 :");                scanf("%d", &mat[i]);
    }
    max_e = max_of(eng, NUMBER);          /* 英语的最高分 */
    max_m = max_of(mat, NUMBER);          /* 数学的最高分 */

    printf("英语的最高分= %d\\n", max_e);
    printf("数学的最高分= %d\\n", max_m);

    return (0);
}
```

运行结果

请输入 5 名学生的分数。

[1] 英语: 53

数学: 82

[2] 英语: 49

数学: 35

[3] 英语: 21

数学: 72

[4] 英语: 91

数学: 35

[5] 英语: 77

数学: 12

英语的最高分 = 91

数学的最高分 = 82

就像之前说明的那样，函数 `max_of` 可以处理任意的数组（当然数组的元素个数也是任意的）。

在本程序中我们使用该函数计算英语和数学分数中的最高分，而其实除了分数之外，例如体重和身高等，只要是 `int` 型的数组都可以处理。

另外，函数 `max_of` 中用来存储分数的数组形参 `vc` 的元素个数是通过接收到的 `no` 来设定的。该函数的函数头如下所示：

```
int max_of(int vc[], int no)
```

其中，`int vc[]` 用来接收数组。函数头也可以写成下面这种形式：

```
int max_of(int *vc, int no)
```

另外，调用函数时使用的实参，只要写明数组的名称就可以了。我们可以像下面这样理解。

在 `main` 函数中传递数组 `eng`（或者 `mat`）给函数 `max_of`，函数 `max_of` 使用名称 `vc` 来接收这个数组。

因此，在函数 `max_of` 中，`vc[0]` 代表 `eng[0]` 的内容，`vc[2]` 代表 `eng[2]` 的内容。由于目前理解起来会比较困难，所以将在第 10 章中介绍该原理。

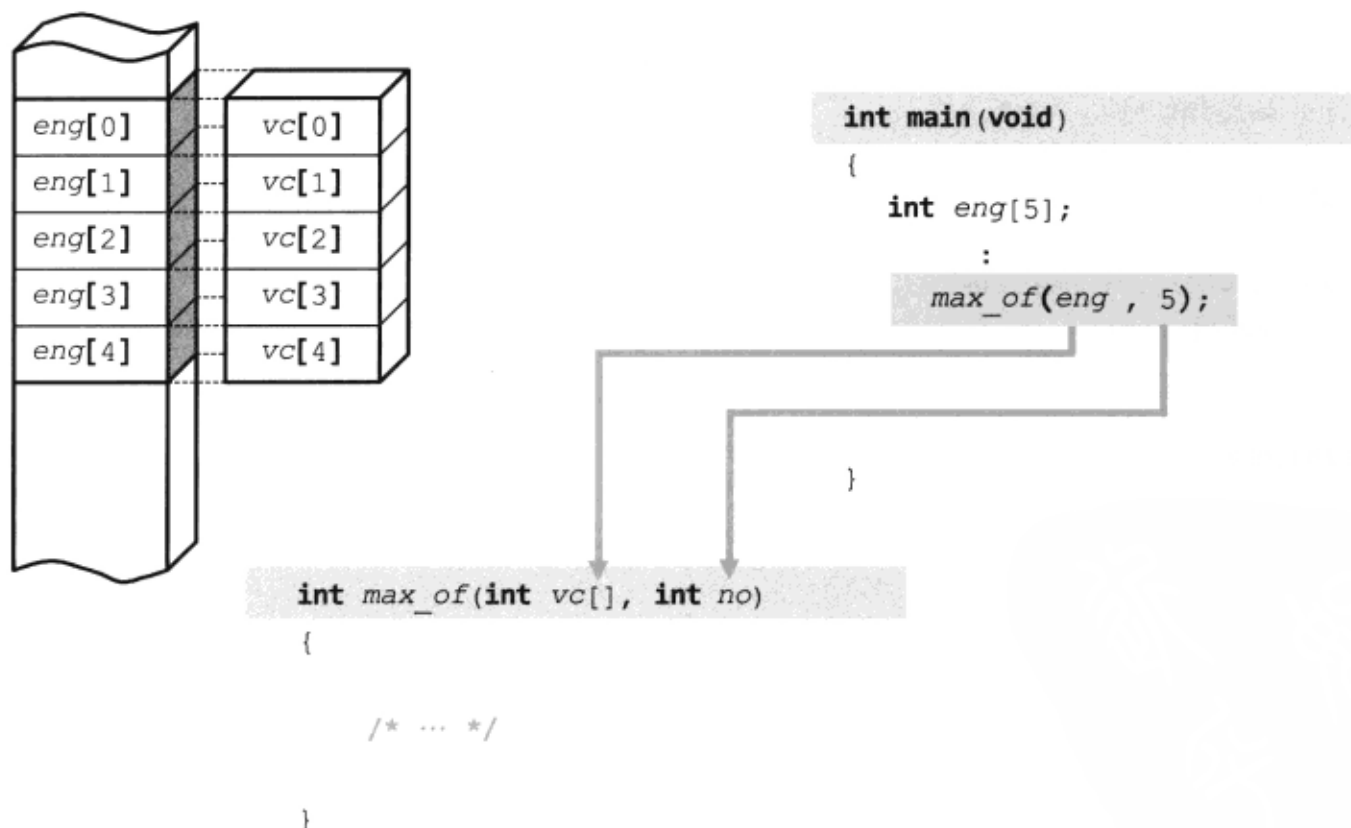


图 6-8 数组的传递

对接收到的数组进行写入处理

通过前面的说明和图例，大家应该已经明白，作为形参接收到的数组，就是调用时被作为实参的数组。或者可以认为这仅仅是给调用函数时的数组赋予了另外一个名称（当然，偶尔也会有实参和形参同名的情况）。

■ 注意 ■

被调用函数中作为形参接收到的数组，就是函数调用时被作为实参的数组。

因此，对接收到的数组元素进行的修改，也会反映到调用时传入的数组中。

利用该原理将 0 赋给数组所有元素的函数如代码清单 6-12 所示。

代码清单 6-12

```

/*
 将数组的所有元素设置为 0
*/

#include <stdio.h>

/*--- 把 0 赋给有 no 个元素的数组 vc 的所有元素 ---*/
void int_set(int vc[], int no)
{
    int i;

    for (i = 0; i < no; i++)
        vc[i] = 0;
}

int main(void)
{
    int i;
    int ary1[] = {1, 2, 3, 4, 5};
    int ary2[] = {3, 2, 1};

    int_set(ary1, 5);          /* 把 0 赋给数组 ary1 的所有元素 */
    int_set(ary2, 3);          /* 把 0 赋给数组 ary2 的所有元素 */

    for (i = 0; i < 5; i++)    printf("ary1[%d] = %d\n", i, ary1[i]);
    for (i = 0; i < 3; i++)    printf("ary2[%d] = %d\n", i, ary2[i]);

    return (0);
}

```

运行结果

```

ary1[0] = 0
ary1[1] = 0
ary1[2] = 0
ary1[3] = 0
ary1[4] = 0
ary2[0] = 0
ary2[1] = 0
ary2[2] = 0

```

const 类型的修饰符

在给函数传递数组的时候，大家可能会担心传递给函数的数组内容会被改变。

为了解决这个问题，C 语言提供了禁止在函数内修改接收到的数组内容的方法。只要在声明形参的时候加上被称为 **const** 的类型修饰符（type qualifier）就可以了。

因此，像代码清单 6-13 那样来实现代码清单 6-11 中的函数的功能更好。这样就可以放心地把数组传递给函数了。

代码清单 6-13

```
/*--- 返回元素个数为 no 的数组 vc 中的最大值 ---*/
int max_of(const int vc[], int no)
{
    int i;
    int max = vc[0];
    for (i = 1; i < no; i++)
        if (vc[i] > max)
            max = vc[i];
    return (max);
}
```

- 如果程序试图修改这样声明出来的数组中的元素，例如 `vc[1] = 5;`，就会在编译的时候发生错误。

● 练习 6-7

创建一个函数，返回元素个数为 *no* 的 **int** 型数组 *vc* 中的最小值。

```
int min_of(const int vc[], int no) { /* ... */ }
```

● 练习 6-8

创建一个函数，对元素个数为 *no* 的 **int** 类型数组 *vc* 进行倒序排列。

```
void rev_intary(int vc[], int no) { /* ... */ }
```

● 练习 6-9

创建一个函数，对元素个数为 *no* 的 **int** 类型数组 *vc* 进行倒序排列，并将其结果保存在数组 *v1* 中。

```
void intary_rcpy (int v1[], const int v2[], int no) { /* ... */ }
```

顺序查找

在数组的元素中查找目标值的程序如代码清单 6-14 所示。函数 `search` 从元素数为 `no` 的 `int` 型数组的开头，顺次查找是否存在与 `key` 值相同的元素。如果有，则返回数组元素的下标。如果没有，则返回 `FAILED`，也就是 -1。

代码清单 6-14

```
/*
    顺序查找
*/

#include <stdio.h>

#define NUMBER      5
#define FAILED      -1

/*--- 查找元素数为 no 的数组 vc 中与 key 一致的元素 ---*/

int search(const int vc[], int key, int no)
{
    int i = 0;

    while (1) {
        if (i == no)
            return (FAILED); /* 查找失败 */
        if (vc[i] == key)
            return (i);      /* 查找成功 */
        i++;
    }
}

int main(void)
{
    int i, ky, idx;
    int vx[NUMBER];

    for (i = 0; i < NUMBER; i++) {
        printf("vx[%d]:", i);
        scanf("%d", &vx[i]);
    }
    printf("要查找的值:");
    scanf("%d", &ky);

    idx = search(vx, ky, NUMBER);

    if (idx == FAILED)
        puts("\a 查找失败。");
    else
        printf("%d 是数组的第 %d 号元素。\\n", ky, idx + 1);

    return (0);
}
```

运行结果 (1)

```
vx[0]: 83
vx[1]: 55
vx[2]: 777
vx[3]: 499
vx[4]: 20
要查找的值: 18
查找失败。
```

运行结果 (2)

```
vx[0]: 83
vx[1]: 55
vx[2]: 777
vx[3]: 499
vx[4]: 20
要查找的值: 499
499 是数组的第 4 号元素。
```


函数 `search` 中 `while` 语句的控制表达式是“1”，因此只有在执行 `return` 语句的时候才能跳出循环，否则循环体将会一直重复执行下去。

在满足下述任意条件的时候，就可以跳出 `while` 语句。

(a) 未能找到想要查找的值，最后跳出循环（当 `i == no` 成立的时候）

(b) 找到了想要查找的值（当 `vc[i] == key` 成立的时候）

该查找过程如图 6-9 所示。

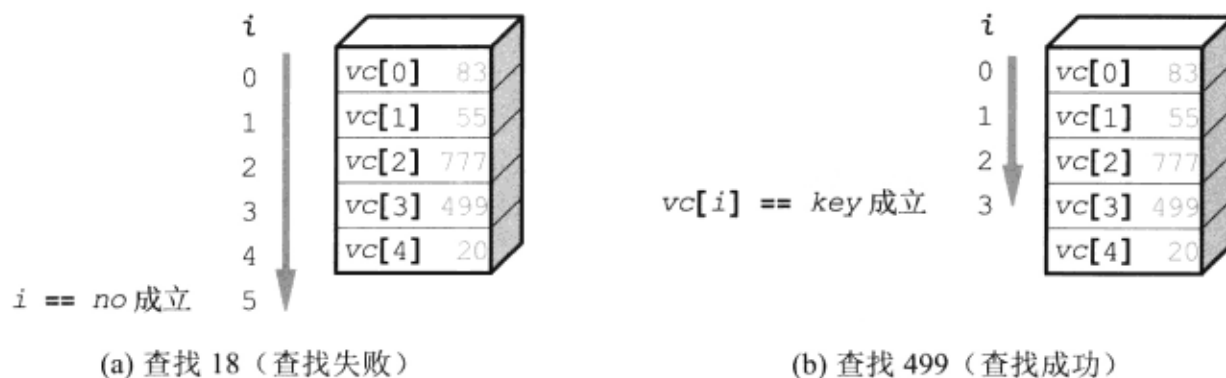


图 6-9 顺序查找

像这样，从数组的开头出发顺次搜索，找出与目标相同的元素的一系列操作，称为**顺序查找** (sequential search)。

哨兵查找法

进行循环操作的时候，需要不停判断是否满足两个结束循环的条件。虽然说判断很简单，但是经过数次累积之后，也是个不小的负担了。

如果数组的大小（元素个数）还有富余，我们就可以把想要查找的数值存储到数组末尾的元素 `vc[no]` 中见图 6-10。

这样一来，即使数组中没有想要查找的数值，当遍历到 `vc[no]` 的时候，也肯定会满足条件 b，这样条件 a 就可以省略了。

在数组末尾追加的数据称为**哨兵** (sentinel)，使用哨兵进行查找的方法称为**哨兵查找法**。使用哨兵可以简化对循环结束条件的判断。

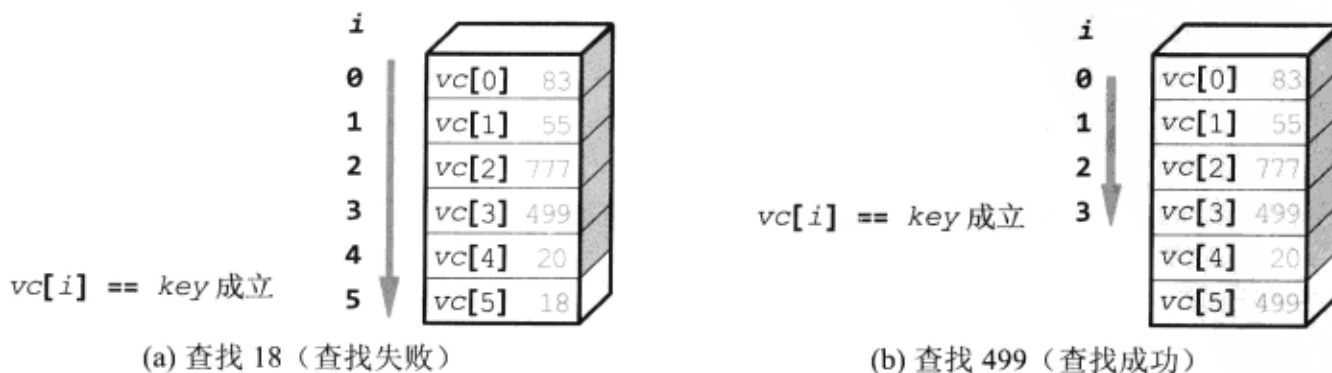


图 6-10 顺序查找（哨兵查找法）

使用哨兵查找法实现顺序查找的程序如代码清单 6-15 所示。

代码清单 6-15

```
/*
    顺序查找（哨兵查找法）
*/

#include <stdio.h>

#define NUMBER      5
#define FAILED      -1

/*--- 通过哨兵查找法顺序查找 ---*/
int search(int vc[], int key, int no)
{
    int i = 0;

    vc[no] = key;      /* 添加哨兵 */

    while (1) {
        if (vc[i] == key)
            break;
        i++;
    }

    return (i == no ? FAILED : i);
}

int main(void)
{
    int i, ky, idx;
    int vx[NUMBER+1];

    for (i = 0; i < NUMBER; i++) {
        printf("vx[%d]:", i);
        scanf("%d", &vx[i]);
    }
    printf("要查找的值:");
    scanf("%d", &ky);

    idx = search(vx, ky, NUMBER);

    if (idx == FAILED)
        puts("\a 查找失败。");
    else
        printf("%d 是数组的第 %d 号元素。\\n", ky, idx + 1);

    return (0);
}
```

运行结果 (1)

```
vx[0]: 83
vx[1]: 55
vx[2]: 777
vx[3]: 499
vx[4]: 20
要查找的值: 18
查找失败。
```

运行结果 (2)

```
vx[0]: 83
vx[1]: 55
vx[2]: 777
vx[3]: 499
vx[4]: 20
要查找的值: 499
499 是数组的第 4 号元素。
```

► 由于函数 `search` 需要改变数组 `vc` 的内容，因此在声明形参 `vc` 的时候不能加入 `const` 修饰符。

表达式语句和空语句

函数 `search` 通过哨兵查找法使循环结束条件减少到了一个，因此程序看上去更加简洁。

现在我们来查看一下代码清单 6-16，函数 `search` 使用 **for** 语句替代 **while** 语句的循环部分，实现了同样的功能。

代码清单 6-16

```
/*--- 通过哨兵查找法顺序查找 ---*/
int search(int vc[], int key, int no)
{
    int i;

    vc[no] = key;      /* 添加哨兵 */

    for (i = 0 ; vc[i] != key; i++)
        ;

    return (i == no ? FAILED : i);
}
```

在该 **for** 语句中，`i` 会不断地自动增加，直到在数组中找到与 `key` 值相同的元素为止。因此，循环体中并不需要执行什么特别的语句。

程序中蓝色底纹的 `;`，是只包含一个分号的空语句（null statement）。

在表达式末尾加上分号组成的语句称为**表达式语句**（2-1 节）。表达式语句的语法结构如图 6-11 所示。

由此看来，空语句其实就是省略了表达式的表达式语句。

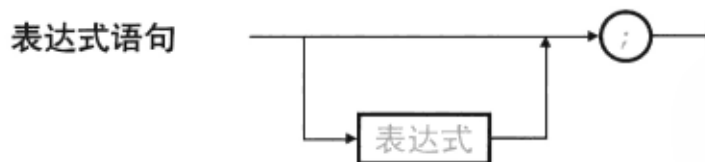


图 6-11 表达式语句的结构图

多维数组的传递

第 5 章给大家介绍了计算两个矩阵之和的程序(5-2 节)。我们要在该程序的基础上,通过独立的函数实现计算矩阵之和功能。程序如代码清单 6-17 所示。

代码清单 6-17

```
/*
    计算 2 行 3 列矩阵之和
*/

#include <stdio.h>

/*--- 把 2 行 3 列的矩阵 ma 和 mb 的和保存到 mc 中 ---*/
void mat_add(const int ma[2][3], const int mb[2][3], int mc[2][3])
{
    int i, j;

    for (i = 0; i < 2; i++)
        for (j = 0; j < 3; j++)
            mc[i][j] = ma[i][j] + mb[i][j];
}

int main(void)
{
    int i, j;
    int ma[2][3] = { {1, 2, 3}, {4, 5, 6} };
    int mb[2][3] = { {6, 3, 4}, {5, 1, 2} };
    int mc[2][3] = { 0 };

    mat_add(ma, mb, mc);          /* 把 ma 和 mb 的和保存到 mc 中 */

    for (i = 0; i < 2; i++) {
        for (j = 0; j < 3; j++)
            printf("%3d", mc[i][j]);
        putchar('\n');
    }

    return (0);
}
```

运行结果

```
7 5 7
9 6 8
```

另外,声明用来接收多维数组的形参的时候,可以省略第一维的元素个数。因此,可以像下面这样声明:

```
void mat_add(const int ma[][3], const int mb[][3], int mc[][3])
```

总而言之，跟一维数组的情况相同，作为形参的数组其实就是作为实参的数组。

► 在三个形参中，只有在声明 *mc* 的时候没有使用 **const** 修饰符，大家应该知道原因吧。

专题 6-1 警告

在调用没有进行函数原型声明的函数时，虽然程序没有语法错误，但是可能会存在某些潜在的错误，大多数编译器在遇到这种情况时会发出警告信息。

● 练习 6-10

创建一个函数，计算 2 行 3 列的矩阵 *ma* 和 3 行 2 列的矩阵 *mb* 的乘积，把结果保存到 2 行 2 列的矩阵 *mc* 中。

```
void mul(const int ma[2][3], const int mb[3][2], int mc[2][2]) { /* ... */ }
```

● 练习 6-11

编写一段程序，使用二维数组操作 5 名学生 3 个科目(语文、数学、英语)的分数，完成如下处理。

例 1：计算每科的最高分。

例 2：计算每名学生 3 个科目的平均分。

6-3 作用域和存储期

作用域和标识符的可见性

在代码清单 6-18 的程序中对变量 *x* 的声明总共有三处。

首先来看一下第一个被声明出来的 *x*。由于它是在函数外面声明定义出来的，所以这个 *x* 拥有文件作用域。

因此，函数 *print_x* 中的 *x* 就是上述的 *x*，屏幕上会输出 *x* = 700。

接下来我们看一下在 **main** 函数中声明、定义出来的 *x*。由于它是在 **main** 函数的程序块也就是复合语句中声明的，所以这个名称在 **main** 函数结尾的大括号 } 之前都是通用的。

代码清单 6-18

```
/*
    确认标识符的作用域
*/

#include <stdio.h>

int x = 700; /* 文件作用域 */

void print_x(void)
{
    printf("x = %d\n", x);
}

int main(void)
{
    int i;
    int x = 800; /* 块作用域 */

    print_x();

    printf("x = %d\n", x);

    for (i = 0; i < 5; i++) {
        int x = i * 100; /* 块作用域 */
        printf("x = %d\n", x);
    }

    printf("x" = %d\n", x);

    return (0);
}
```

运行结果

```
x = 700
x = 800
x = 0
x = 100
x = 200
x = 300
x = 400
x = 800
```

如果两个同名变量分别拥有文件作用域和块作用域，那么只有拥有块作用域的变量是“可见”的，而拥有文件作用域的变量会被“隐藏”起来。

变量的可见性请参考图 6-12。

因此，在 **main** 函数中调用第一个 **printf** 函数的时候，*x* 的值显示为 *x* = 800。

在 **main** 函数的 **for** 语句中声明定义了第三个变量 *x*。当同名变量都被赋予了块作用域的时候，内层的变量是“可见”的，而外层的变量会被“隐藏”起来。

综上所述，**for** 语句循环体这个程序块中的 *x* 实际上就是上述第三个变量 *x*。由于 **for** 语句的循环执行了 5 次，所以显示出的 *x* 的值如下：

```
x = 0
x = 100
x = 200
x = 300
x = 400
```

for 语句的循环结束之后，该变量 *x* 的名称就会失效。因此，在调用最后一个 **printf** 函数的时候，显示出的 *x* 的值是 *x* = 800。

► 从图 6-12 中可以看出，被声明的标识符从其名称书写出来之后生效。

因此，即使把对 *x* 的声明修改为 **int x = x;**，作为初始值的 *x* 也是被声明出来的 *x*，而不是拥有文件作用域的那个 *x*。它的初始值不是 700，而是被初始化为不确定的值（请参考后续内容）。

```
/*
    确认标识符的作用域
*/
#include <stdio.h>

int x = 700;

void print_x(void)
{
    printf("x = %d\n", x);
}

int main(void)
{
    int i;
    int x = 800;

    print_x();

    printf("x = %d\n", x);

    for (i = 0; i < 5; i++) {
        int x = i * 100;
        printf("x = %d\n", x);
    }

    printf("x = %d\n", x);

    return (0);
}
```

图 6-12 标识符的可见性

存储期

在函数中声明的变量，并不是从程序开始到程序结束始终有效的。变量的生存期也就是寿命有两种，它们可以通过**存储期**（storage duration）这个概念来体现。

下面就通过代码清单 6-19 中的程序来具体说明。

在函数 *func* 中声明了 *sx* 和 *ax* 两个变量。但是，声明 *sx* 的时候我们使用了**存储类说明符 static**。

正因为如此，最终 *ax* 和 *sx* 的值并不相同，我们可以通过程序的运行结果来确认。程序中的注释也对它们拥有不同的存储类型进行了说明。

■ 自动存储期（automatic storage duration）

在函数中不使用存储类说明符 **static** 而定义出的对象（变量），被赋予了自动存储空间，它具有以下特性：

程序执行到对象声明的时候就创建出了相应的对象。而执行到包含该声明的程序块的结尾，也就是大括号 `}` 的时候，该对象就会消失。

也就是说，该对象拥有短暂的寿命，仅在大括号包围的程序块中才能生存。

被赋予自动存储期的对象，在程序执行到 `int ax = 0;` 的时候，就被创建出来并且进行初始化。

另外，如果不显式地进行初始化，则该对象会被初始化为不确定的值。

■ 静态存储期（static storage duration）

在函数中使用 **static** 定义出的对象，或者是在函数外声明定义出的对象都被赋予了静态存储期，它具有以下特性：

在程序开始执行的时候，具体地说是在 **main** 函数执行之前的准备阶段被创建出来，在程序结束的时候消失。

也就是说，该对象拥有“永久”的寿命。

被赋予了静态存储空间的对象，会在 **main** 函数开始执行之前被初始化。因此，虽说程序执行的时候会经过 `static int sx = 0;`，但其实那个时候并没有进行初始化处理，也就是说

该声明并未执行赋值处理。

另外，如果没有赋予初始值，则该对象会自动初始化为 0。

代码清单 6-19

```
/*
    自动存储期和静态存储期
*/

#include <stdio.h>

int fx = 0; /* 静态存储期，文件作用域 */

void func(void)
{
    static int sx = 0; /* 静态存储期，块作用域 */
    int ax = 0; /* 自动存储期，块作用域 */

    printf("%3d%3d%3d\n", ax++, sx++, fx++);
}

int main(void)
{
    int i;

    puts(" ax sx fx");
    puts("-----");
    for (i = 0; i < 10; i++)
        func();
    puts("-----");

    return (0);
}
```

运行结果

ax	sx	fx
0	0	0
0	1	1
0	2	2
0	3	3
0	4	4
0	5	5
0	6	6
0	7	7
0	8	8
0	9	9

► 在函数中通过存储类说明符 **auto** 或者 **register** 声明定义出的变量，也被赋予了自动存储期。通过 **auto int ax = 0;** 进行的声明和不使用 **auto** 进行的声明在编译的时候是完全相同的。因此 **auto** 就显得有些多余了。

另外，使用 **register** 进行的声明 **register int ax = 0;**，在源程序编译的时候，变量 **ax** 不是保存在内存中，而是保存在更高速的寄存器中。然而，由于寄存器的数量有限，所以也不是绝对的。

现在的编译技术已经十分先进了，哪个变量保存在寄存器中更好都是通过编译自行判断并进行最优化处理的（不仅如此，保存在寄存器中的变量在程序执行的时候也可能发生改变）。

使用 **register** 进行声明也渐渐变得没有意义了。

在理解以上两个存储期的含义的基础之上，我们来通过图 6-13 研究一下程序的处理流程。

► 该图中蓝色底纹部分被赋予了静态存储期的变量，灰色底纹部分表示被赋予了自动存储期的变量。

(a) 拥有静态存储期的变量 fx 和 sx ，在程序开始的时候（**main** 函数执行之前）被创建出来，并被初始化为 0。

(b) 当 **main** 函数开始执行的时候，创建出了变量 i 。

(c) 在 **main** 函数中调用函数 $func$ 的时候，创建了变量 ax 并将其初始化为 0。这样，变量 ax 、 sx 、 fx 的值分别是 0 0 0。之后这三个变量全都会自动增加为 1。

(d) 当函数 $func$ 执行结束的时候变量 ax 就消失了。

(e) **main** 函数中的变量 i 自动增加，然后再调用函数 $func$ 。这时，变量 ax 再次被创建出来并被初始化为 0。于是这三个变量的值分别为 0 1 1。在显示处理结束之后，这些变量的值自动增加为 1、2、2。

main 函数总共调用了 10 次函数 $func$ ，拥有“永久”寿命的变量 fx 和 sx 会一直自动增加。而只存在于函数 $func$ 中的变量 x ，由于每次创建的时候都被初始化为 0，因此被创建了 10 次之后，它的值还是 0。

(f) **main** 函数执行结束的同时，变量 i 也会消失。

我们可以通过代码清单 6-20 中的程序，来确认拥有静态存储期的对象是否会被自动初始化为 0。

● 练习 6-12

编写一段程序，为数组中的全部元素分配静态存储期，并确认它们都被初始化为 0。

第 7 章

基本数据类型

`int` 型是一种只表示整数的数据类型，它不能表示具有小数部分的实数。所以我们在前几章中是用 `double` 型来处理实数的。

由此可见，数值表现都有一定的特征和范围，它们是由数据类型决定的。C 语言提供了丰富的数据类型。本章我们就来学习这些基本数据类型。

7-1 基本数据类型和数

基本数据类型

经过前几章的学习，我们知道可以对 **int** 型和 **double** 型的变量及常量进行加减等算术运算。这种数据类型称为**算术类型**（arithmetic type）。

算术类型如图 7-1 所示，是多种数据类型的统称，大体上可分为以下两种类型。

整数类数据类型（integral type）：只表示整数。

浮点型（floating type）：可表示具有小数部分的数值。

前者（整数类数据类型）是以下数据类型的统称。

枚举型（enumeration type）※ 在下一章学习

字符型（character type）：表示字符

整型（integer type）：表示整数

字符型、整型和浮点型只需使用 **int** 或 **double** 等关键字就能表示其数据类型，因此将它们统称为**基本数据类型**（basic type）。

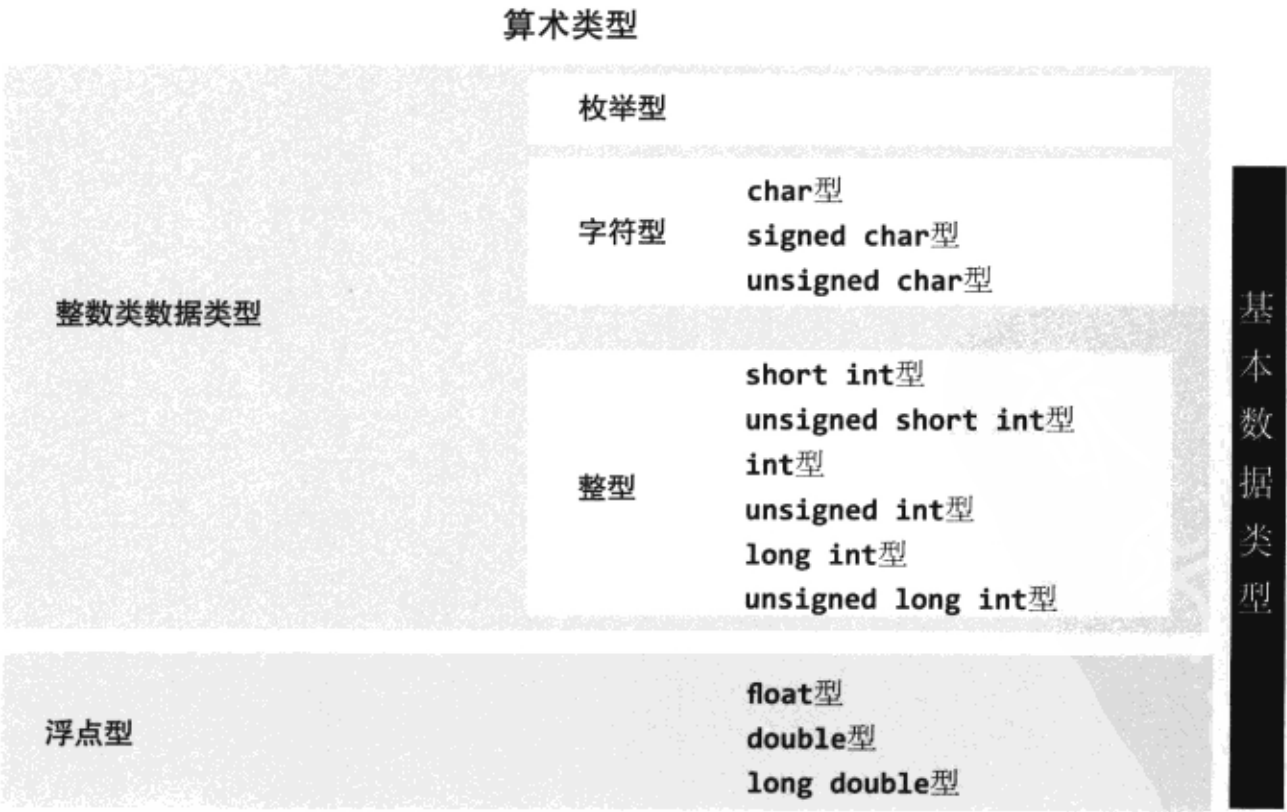


图 7-1 算术类型

基数

我们先来学习整数。

我生于 1963 年。这种数值表现形式很常见，众所周知这是以 10 为**基数**的**十进制数**。

不过，在大家使用的电子计算机中所有数据都是用 ON/OFF 信号（即 1 和 0）来表示的。对我们来说最容易理解的是十进制数，而对计算机来说以 2 为基数的**二进制数**则更易于理解。

虽说如此，假如我们将所有数值都用二进制数来表示可就太费力劳神了。如果只能使用二进制数，那么在自我介绍时必须得说“我生于 11110101011 年”了。

且不说这样的数值如何，就接近硬件底层的程序来说，使用二进制数会更加适宜。二进制数固然有其优点，却也存在位数过多处理不便的问题，所以在写法上还使用了**八进制数**和**十六进制数**。

在十进制数中，如果以下 10 种数字都用完了，就进位为 10。

0 1 2 3 4 5 6 7 8 9 1 位十进制数

同样在八进制数中，用完以下 8 种数字后就进位为 10。

0 1 2 3 4 5 6 7 1 位八进制数

以此类推，在十六进制数中使用以下 16 种数字，那么 F 后面的数就是 10。

0 1 2 3 4 5 6 7 8 9 A B C D E F 1 位十六进制数

如下所示，将十进制数 0 ~ 18 分别用八进制、十进制和十六进制表示。

八进制数	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20	21	22	...
十进制数	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...
十六进制数	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	...

二进制只使用 0 和 1 两种数字表示数值。

0 1 1 位二进制数

因此，将十进制的 0 ~ 12 用二进制表示就是：

0 1 10 11 100 101 110 111 1000 1001 1010 1011 1100

基数转换

■ 由八进制数、十六进制数、二进制数向十进制数转换

十进制数的每一位都是 10 的指数幂。所以 1998 可以解释为

$$1998 = \underset{1000}{1 \times 10^3} + \underset{100}{9 \times 10^2} + \underset{10}{9 \times 10^1} + \underset{1}{8 \times 10^0}$$

将这个思路应用于八进制数、十六进制数和二进制数上，就能轻松地将这些数转换为十进制数。

举例来说，将八进制数 123 转换为十进制数的步骤如下：

$$\begin{aligned} 123 &= 1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 \\ &= 1 \times 64 + 2 \times 8 + 3 \times 1 \\ &= 83 \end{aligned}$$

而将十六进制数 1FD 转换为十进制数的步骤如下：

$$\begin{aligned} 1FD &= 1 \times 16^2 + 15 \times 16^1 + 13 \times 16^0 \\ &= 1 \times 256 + 15 \times 16 + 13 \times 1 \\ &= 509 \end{aligned}$$

将二进制数 101 转换为十进制数的步骤如下：

$$\begin{aligned} 101 &= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 4 + 0 \times 2 + 1 \times 1 \\ &= 5 \end{aligned}$$

■ 由十进制数向八进制数、十六进制数、二进制数转换

二进制数有以下规律。

偶数的末位数字为 0。

奇数的末位数字为 1。

也就是说，用要转换的数除以 2 所得的余数就是末位数字的值。

例如，十进制数 57 除以 2 的余数为 1，那么转换后的二进制数的末位数字就是 1，这一点只要稍作计算就能明白了。

在继续十进制数转二进制数的话题之前，我们先对“十进制数转换为十进制数”的方法作一下说明。一个数除以 10 的余数，与这个数的末尾数字相等。例如，1962 除以 10 的余数为 2，

与末位数字 2 相等。

此处除法运算 $1962/10$ 的商为 196，也就是 1962 右移 1 位后的值（删去末位的 2）。即十进制数除以 10 的意思就是右移 1 位。接着用 196 除以 10，得到的余数 6 就是倒数第 2 位的值。继续将此时的商 19 除以 10……

将一个数除以 10，求得商和余数，再对商作同样的除法计算。重复这一过程，直到商为 0 为止，最后将求得的所有余数逆向排列，就得到了转换后的十进制数（图 7-2）。



图 7-2 将十进制数 1962 转换为十进制数



图 7-3 将十进制数 57 转换为二进制数

在上述步骤中将 10 改为 2 就是“十进制数转二进制数”的方法了。因为用一个数除以 2 就相当于将它的二进制数右移 1 位。

现在我们回到将十进制数 57 转换为二进制数的话题。用 57 除以 2，商为 28，余数为 1。再用商 28 除以 2，得到商 14 和余数 0。反复这一步骤，直到商为 0 为止，将所有余数逆向排列就得到了结果 111001（图 7-3）。

当然，转换为十进制数、八进制数、十六进制数的步骤是完全相同的。只要将除数改为 8 或 16，最后排列余数就行了。

十进制数 57 转换为八进制数为 71（图 7-4），转换为十六进制数为 39（图 7-5）。

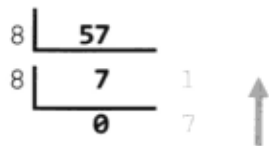


图 7-4 将十进制数 57 转换为八进制数

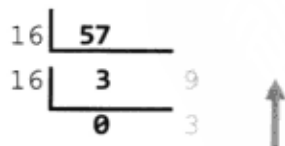


图 7-5 将十进制数 57 转换为十六进制数

① C99标准还定义了long long类型，其长度可以保证至少64位。

图 7-6 是这些数据类型的汇总。

► 与 **signed** 和 **unsigned** 相同，**short** 和 **long** 也是一种类型说明符。

各种数据类型具体能表示多少个数值因编译器而异。本书设定这些数据类型可表示的数值范围如下（专题 7-1）。

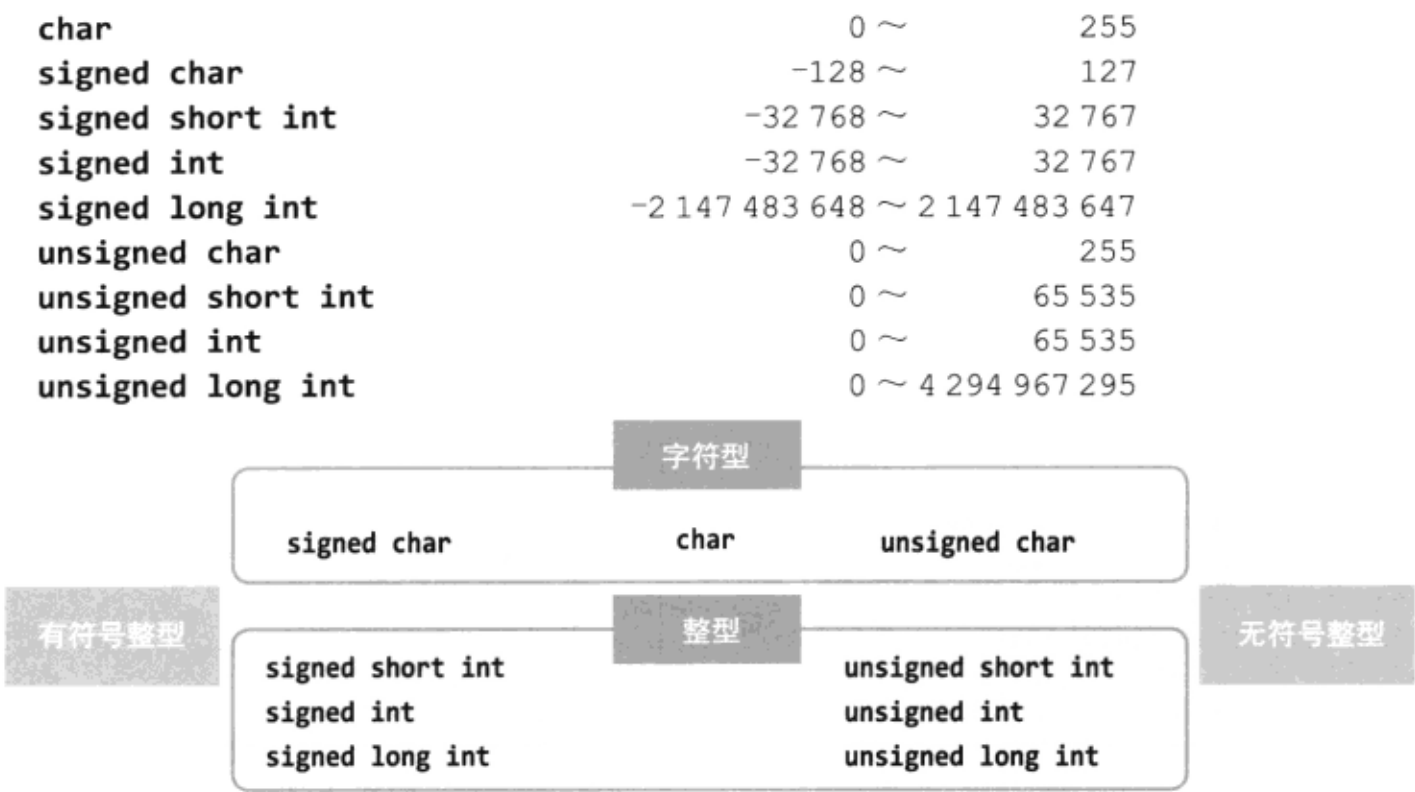


图 7-6 表示整数的数据类型分类

专题 7-1 字符型和整型的表示范围

字符型和整型至少可以表示以下范围的整数。

char	0 ~ 255 或者 -127 ~ 127
signed char	-127 ~ 127
signed short int	-32 767 ~ 32 767
signed int	-32 767 ~ 32 767
signed long int	-2 147 483 647 ~ 2 147 483 647
unsigned char	0 ~ 255
unsigned short int	0 ~ 65 535
unsigned int	0 ~ 65 535
unsigned long int	0 ~ 4 294 967 295

以上是本章设定的数值范围，实际上在许多编译器中可以多表示 1 个负数。例如 **signed short int** 型的表示范围为 -32768 至 32767。这是因为使用了补码，在后文会讲到。而在使用反码的编译器中，所能表示的整数的正负范围相同。

<limits.h> 头文件

C 语言编译器在 <limits.h> 头文件中以宏定义的形式定义了字符型以及其他整型所能表示的数值的最小值和最大值。

本书设定编译器中 <limits.h> 的内容如代码清单 7-1 所示。

代码清单 7-1

```

/*
 * 本书设定的 <limits.h> 的部分内容
 */

#define UCHAR_MAX      255U          /* unsigned char 的最大值 */
#define SCHAR_MIN      -128          /* signed char 的最小值 */
#define SCHAR_MAX      +127          /* signed char 的最大值 */
#define CHAR_MIN0      /* char 的最小值 */
#define CHAR_MAX UCHAR_MAX          /* char 的最大值 */
#define SHRT_MIN -32768              /* short int 的最小值 */
#define SHRT_MAX +32767              /* short int 的最大值 */
#define USHRT_MAX      65535U        /* unsigned short 的最大值 */
#define INT_MIN         -32768        /* int 的最小值 */
#define INT_MAX          +32767       /* int 的最大值 */
#define UINT_MAX 65535U              /* unsigned int 的最大值 */
#define LONG_MIN -2147483648L         /* long int 的最小值 */
#define LONG_MAX +2147483647L         /* long int 的最大值 */
#define ULONG_MAX      4294967295UL   /* unsigned long 的最大值 */

```

代码清单 7-2 所示的程序显示了由这个宏定义规定的 **char** 型、**short int** 型、**int** 型和 **long int** 型可表示的数值范围。

代码清单 7-2

```

/*
 * 显示各种整数数据类型的表示范围
 */

```

```

#include <stdio.h>
#include <limits.h>

```

```
int main(void)
{

```

```

    printf("char      : %d ~ %d\n", CHAR_MIN, CHAR_MAX);
    printf("short int : %d ~ %d\n", SHRT_MIN, SHRT_MAX);
    printf("int       : %d ~ %d\n", INT_MIN, INT_MAX);
    printf("long int  : %ld ~ %ld\n", LONG_MIN, LONG_MAX);

```

```

    return (0);
}

```

运行结果

```

char      : 0~255
short int : -32768~32767
int       : -32768~32767
long int  : -2147483648~2147483647

```

► 运行结果因编译器和运行环境或有不同。

"%ld" 转换说明表示 **long int** 型数值。请在你所使用的编译器中运行以上程序确认运行结果。

字符型

char 型是用来保存“字符”的数据类型。

对于没有声明 **signed** 和 **unsigned** 的 **char** 型，视为有符号类型还是无符号类型，由编译器而定。

代码清单 7-3 所示程序判断了 **char** 型是否有符号，并显示出了结果。

代码清单 7-3

```
/*
  判断char型有无符号
*/

#include <stdio.h>
#include <limits.h>

int main(void)
{
    printf("这个编译器中的 char 型是");

    if (CHAR_MIN)
        puts("有符号的。");
    else
        puts("无符号的。");

    return (0);
}
```

运行结果

这个编译器中的char型是无符号的。

► 运行结果因编译器和运行环境或有不同。

在 **char** 型为无符号类型的编译器中，<limit.h> 中的定义如下：

```
#define CHAR_MIN 0
#define CHAR_MAX UCHAR_MAX          /* 与 unsigned char 的最大值相同 */
```

在 **char** 型为有符号类型的编译器中，定义如下：

```
#define CHAR_MIN SCHAR_MIN          /* 与 signed char 的最小值相同 */
#define CHAR_MAX SCHAR_MAX          /* 与 signed char 的最大值相同 */
```

因此，在这个程序中，通过对 **CHAR_MIN** 的值是否为 0 来判断 **char** 的类型。

本书假定 **char** 型为无符号数据类型。

► 下一章也会学习有关字符的内容。

sizeof 运算符

C 语言中将表示字符的 **char** 型的长度定义为 1。

通过使用 **sizeof 运算符** (sizeof operator)，写作 **sizeof(数据类型名称)** 可以判断出包括 **char** 型在内的所有数据类型的长度。

► 本章最后还会介绍 **sizeof** 运算符的相关内容。

代码清单 7-4 所示的程序显示了几种数据类型的长度。

代码清单 7-4

```
/*
 * 显示数据类型长度
 */
#include <stdio.h>

int main(void)
{
    printf("sizeof(char) = %u\n", (unsigned)sizeof(char));
    printf("sizeof(short) = %u\n", (unsigned)sizeof(short));
    printf("sizeof(int) = %u\n", (unsigned)sizeof(int));
    printf("sizeof(long) = %u\n", (unsigned)sizeof(long));

    return (0);
}
```

运行结果

```
sizeof(char) = 1
sizeof(short) = 2
sizeof(int) = 2
sizeof(long) = 4
```

► 运行结果因编译器和运行环境或有不同。

在这个程序中，分别将 **short int** 型和 **long int** 型写作 **short** 和 **long**。单独出现 **short** 和 **long** 的时候，视为省略 **int** 关键字。同样，将单独出现的 **signed** 和 **unsigned** 单纯地视为 **int** 型（而非 **short** 型或 **long** 型）。

表 7-1 归纳了上述关系。表中的行表示同一种数据类型。例如，**signed long int**、**signed long**、**long int**、**long** 都是相同的数据类型。本书使用了最简短的表示方式，即表的最右侧一列蓝色底纹部分所示的数据类型名称。

■ 表 7-1 字符型、整型的数据类型名称

字符型	char			
	signed char			
	unsigned char			
整型	signed short int	signed short	short int	short
	unsigned short int	unsigned short		
	signed int	signed	int	
	unsigned int	unsigned		
	signed long int	signed long	long int	long
	unsigned long int	unsigned long		

size_t 型和 typedef 声明

由 **sizeof** 运算符生成的值的数据类型是在 `<stddef.h>` 头文件中定义的 **size_t** 型。在许多编译器中用 **typedef** 声明来定义 **size_t** 型。

```
typedef unsigned size_t;
```

typedef 声明是数据类型的同义词，也就是为现有的数据类型创建别名（并非创建新的数据类型）。通过这个声明，**size_t** 就成了 **unsigned** 型的同义词。这个同义词在语法上可以作为一种“数据类型名称”来使用。

sizeof 运算符是不会生成负值的，所以将它定义为无符号整型。

不过有些编译器可能会将 **size_t** 型定义为 **unsigned short** 型或 **unsigned long** 型的同义词。

print 函数中的 `"%lu"` 转换说明表示 **unsigned long** 型数值，这在本书 5-3 节中已有说明。`"%u"` 转换说明则是表示 **unsigned** 型数值。

格式控制字符串中的转换说明必须和要显示的值的数据类型保持一致。因此在显示 **size_t** 型的值时，必须要像上述程序那样将它们“对号入座”。

► 当然，以下语句也是可行的。

```
printf( "sizeof(int)   = %lu\n ", (unsigned long)sizeof(int));
```

整型的灵活运用

通常，**int** 型是程序运行环境中最易处理并且可以进行高速运算的数据类型。在有些 **sizeof(long)** 大于 **sizeof(int)** 的编译器中，**long** 型的运算比 **int** 型更耗时。因此只要我们不处理特别大的数值，还是尽量使用 **int** 型或 **short** 型比较好。

► 各种数据类型的有符号版和无符号版的长度相同（例如，**sizeof(short)** 和 **sizeof(unsigned short)** 相等）。**short**、**int** 和 **long** 具有以下关系：

```
sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
```

其中，右边的数据类型是大于还是等于左边的数据类型，是根据编译器而定的。

因此，我们要注意在有些编译器中这 3 种数据类型的长度可能完全相等。本书假定如下。

```
sizeof(short) = sizeof(int) = 2
```

```
sizeof(long)  = 4
```

整型常量

整型常量的语法结构如图 7-7 所示。

■ 十进制常量

我们使用的 10、57 等整型常量称为**十进制常量** (decimal constant)。

■ 八进制常量

八进制常量 (octal constant) 以 0 开头，以区别于十进制常量。以下两个整型常量看似相同，但实际上它们的值完全不同。

12 —— 十进制常量 (十进制的 12)

012 —— 八进制常量 (十进制的 10)

■ 十六进制常量

十六进制常量 (hexadecimal constant) 以 0x 或 0X 开头。A ~ F 不区分大小写。示例如下：

0xA —— 十六进制常量 (十进制的 10)

0x12 —— 十六进制常量 (十进制的 18)

整型常量的数据类型

请看代码清单 7-1 所示的程序。在几个常量的末尾都附带了 U、L、UL 等符号。这些符号是**整型后缀** (integer suffix)，可以为整型常量加上 u 或者 U 表明无符号类型，也可以加上 l 或者 L 表明 **long** 型。例如可采用下述写法：

```
38000U      /* 无符号类型 */
120000L     /* long 型 */
```

► 小写字母 l 和数字 1 很容易混淆，推荐使用大写字母 L。

由此可知，整型常量的数据类型不限于 **int** 型。

对于 1000、60000 等不带整型后缀的十进制数，如果该值在 **int** 型可表示的数值范围之内，那么它的数据类型就为 **int** 型。否则，如果在 **long** 型可表示的数值范围之内，就是 **long** 型。除此之外都是 **unsigned long** 型。

上述规则归纳如下：

(a) 无后缀的十进制常量	int →	long → unsigned long
(b) 无后缀的八进制或十六进制常量	int → unsigned →	long → unsigned long
(c) 带后缀 u/U	unsigned →	unsigned long
(d) 带后缀 l/L		long → unsigned long
(e) 带后缀 l/L 和 u/U		unsigned long

举例如下（各种数据类型以本书设定的表示范围为例）：

1000 （a）类。能用 **int** 型表示，所以为 **int** 型。

60000 （a）类。不能用 **int** 型表示，但能用 **long** 型表示，所以为 **long** 型。

60000U （c）类。能用 **unsigned** 型表示，所以为 **unsigned** 型。

60000 和 60000U 的数值虽然相同，却分别是 **long** 型和 **unsigned** 型。由此可见，数值相同的常量也会因为有无整型后缀而成为不同的数据类型。

因此我们应该根据需要加上适当的整型后缀。

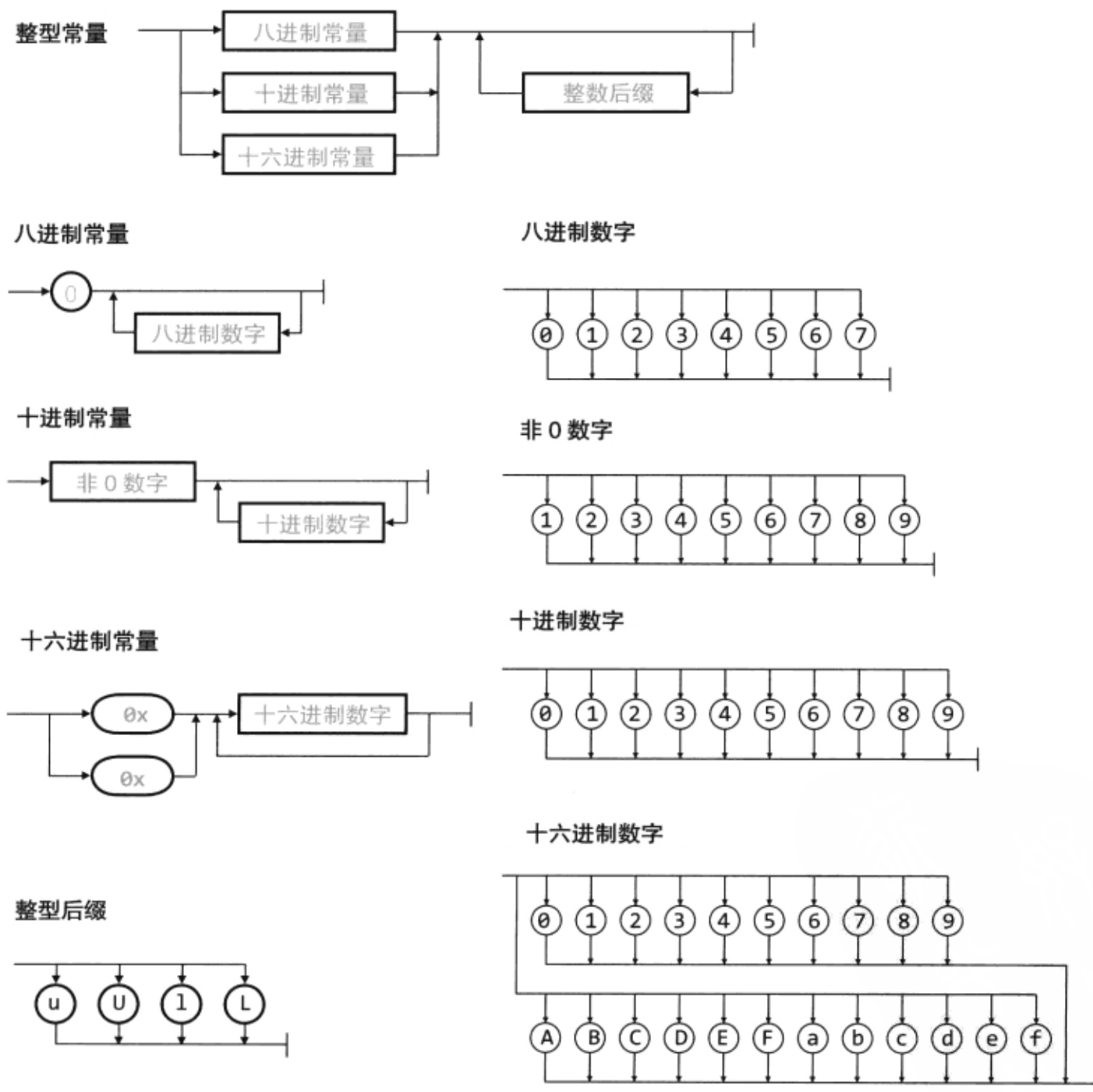


图 7-7 整型常量的语法结构图

► 整型后缀 U (u) 和 L (l) 的顺序任意。所以 5000UL 和 5000LU 是完全一样的。

内部表示和位

我们一直将变量当做保存数值的魔术盒。计算机中的所有数据都是用 0 和 1（即“位”）的组合来表示的。所以在盒子的内部也是以 0 和 1 的位序列来表示数据的。

那么整型数据的内容是怎么样的呢？

► 以下是“位”在 C 语言中的定义：

“位”是具有大量内存空间的运行环境的数据存储单元，可保存具有两种取值的对象。对象中各二进制位的地址不需要表示。

※ “位”可取两种值，其中一种是 0。将位设为 0 以外的值，称为“设置位”。

无符号整数的内部表示

无符号整数的数值在计算机内部是以二进制数来表示的，该二进制数与各二进制位一一对应。

例如，23 的二进制数是 10111，所以在 unsigned 型为 16 位的编译器中，高位补 0 后表示为 0000000000010111。

依此类推，在 unsigned 型为 16 位的编译器中的表示形式如图 7-8 所示。可以表示 0 到 65535 共 65536（即 2^{16} ）个数字。

► n 位可以表示的无符号整数有 $0 \sim 2^n - 1$ 共 2^n 种。这和 n 位十进制数可表示的数值范围为 $0 \sim 10^n - 1$ 共 10^n 个数字的道理一样（例如，3 位十进制数可表示 $0 \sim 999$ 共 1000 个数字）。

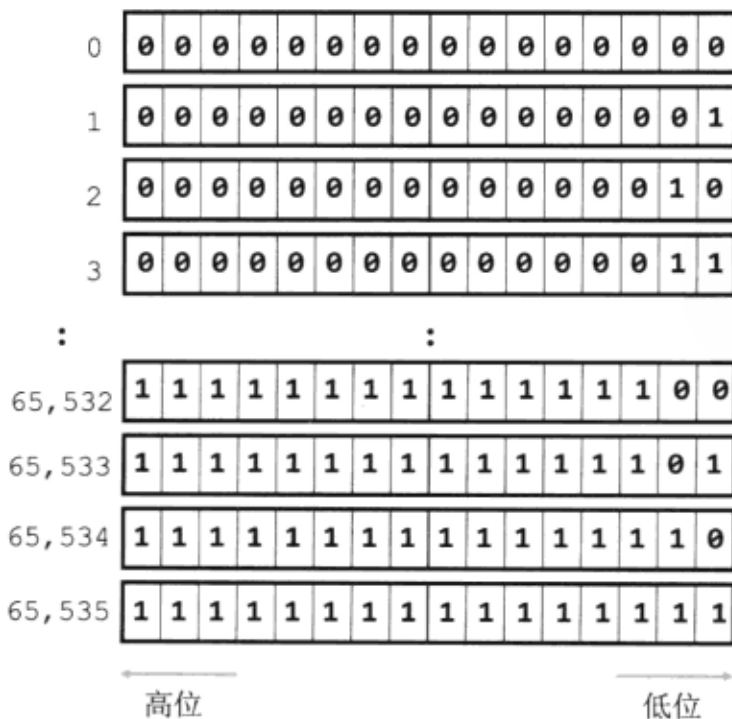


图 7-8 无符号整数的内部表示

有符号整数的内部表示

有符号整数具有多种表示法。这里介绍一下最常用的二进制补码 (two's complement) 表示法。如图 7-9 所示, 最高位 (符号位) 为 1, 表示负数。最高位 (符号位) 为 0, 表示非负数 (0 或正数)。

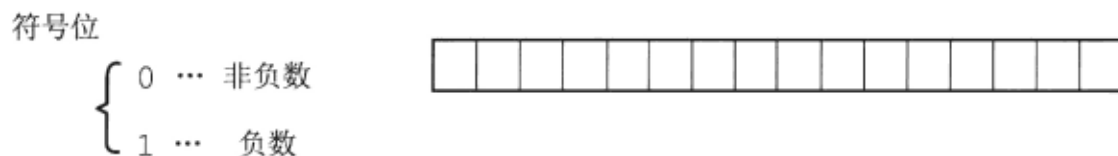


图 7-9 有符号整数和符号位

由于第一位被符号位占去了, 具体数值就用剩下的 15 位来表示。请看图 7-10 中各种数值的表示形式。

非负数有 0 ~ 32767 共 32768 个, 负数有 -1 ~ -32768 共 32768 个。加起来一共可以表示 65536 个数。

- 有符号 `int` 型和无符号 `int` 型的共同部分 0 ~ 32767, 数值表示 (位串) 是相同的。

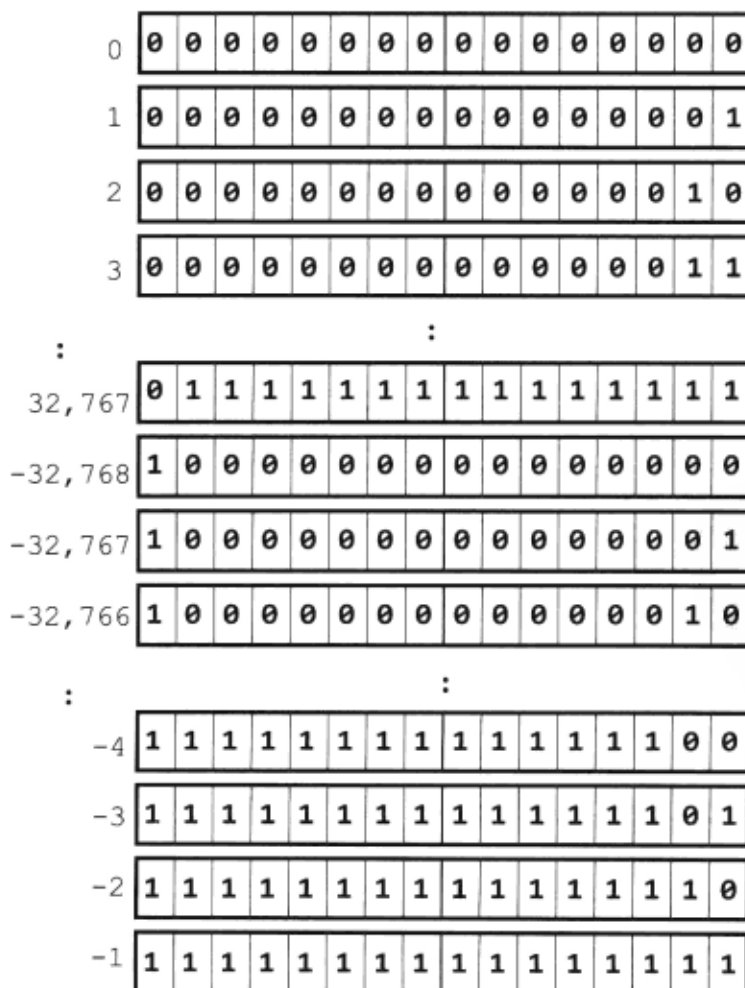


图 7-10 有符号整数的内部表示 (补码)

反码表示法和补码表示法

假设整数有 5 位，我们来对比一下用反码表示法和补码表示法表示负数时，带符号整数的位串有何区别。请看图 7-11。

共同点

- 正数部分的位串相同。
- 负数的最高位都是 1。

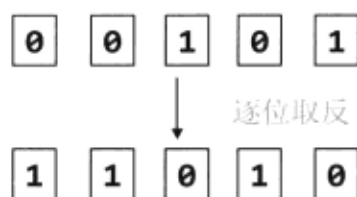
不同点

- 补码表示法可表示 32 个整数，而反码表示法可表示 31 个整数。

反码的求法

求一个正数的反码，需将其逐位取反。

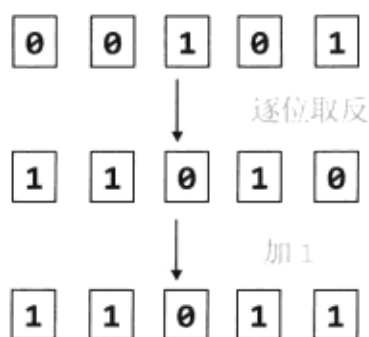
例如，由 5 求出 -5 的步骤如下。



补码的求法

求一个正数的补码，需将其逐位取反后加 1。

例如，由 5 求出 -5 的步骤如下。



	反码	补码
0	0 0 0 0 0	0 0 0 0 0
1	0 0 0 0 1	0 0 0 0 1
2	0 0 0 1 0	0 0 0 1 0
3	0 0 0 1 1	0 0 0 1 1
4	0 0 1 0 0	0 0 1 0 0
5	0 0 1 0 1	0 0 1 0 1
6	0 0 1 1 0	0 0 1 1 0
7	0 0 1 1 1	0 0 1 1 1
8	0 1 0 0 0	0 1 0 0 0
9	0 1 0 0 1	0 1 0 0 1
10	0 1 0 1 0	0 1 0 1 0
11	0 1 0 1 1	0 1 0 1 1
12	0 1 1 0 0	0 1 1 0 0
13	0 1 1 0 1	0 1 1 0 1
14	0 1 1 1 0	0 1 1 1 0
15	0 1 1 1 1	0 1 1 1 1
-16		1 0 0 0 0
-15	1 0 0 0 0	1 0 0 0 1
-14	1 0 0 0 1	1 0 0 1 0
-13	1 0 0 1 0	1 0 0 1 1
-12	1 0 0 1 1	1 0 1 0 0
-11	1 0 1 0 0	1 0 1 0 1
-10	1 0 1 0 1	1 0 1 1 0
-9	1 0 1 1 0	1 0 1 1 1
-8	1 0 1 1 1	1 1 0 0 0
-7	1 1 0 0 0	1 1 0 0 1
-6	1 1 0 0 1	1 1 0 1 0
-5	1 1 0 1 0	1 1 0 1 1
-4	1 1 0 1 1	1 1 1 0 0
-3	1 1 1 0 0	1 1 1 0 1
-2	1 1 1 0 1	1 1 1 1 0
-1	1 1 1 1 0	1 1 1 1 1

图 7-11 反码和补码

窥探整数内部

代码清单 7-5 所示的程序显示了 **unsigned** 型内部的位串。

这个程序中有很多新出现的运算符。它们都是操作整数内部的位的运算符。我们先学习一些位操作的相关知识，然后再去理解程序。

代码清单 7-5

```
/*
 * 显示unsigned型的位串
 */
#include <stdio.h>

/*---返回在整数x中设置的位数---*/
int count_bits(unsigned x)
{
    int count = 0;
    while (x) {
        if (x & 1U) count++;
        x >>= 1;
    }
    return (count);
}

/*---返回unsigned型的位数---*/
int int_bits(void)
{
    return (count_bits(~0U));
}

/*---显示unsigned型的位串内容---*/
void print_bits(unsigned x)
{
    int i;
    for (i = int_bits() - 1; i >= 0; i--)
        putchar(((x >> i) & 1U) ? '1' : '0');
}

int main(void)
{
    unsigned nx;

    printf("请输入一个非负整数: ");
    scanf("%u", &nx);

    print_bits(nx);
    putchar('\n');

    return (0);
}
```

运行结果 (1)

请输入一个非负整数: 0

0000000000000000

运行结果 (2)

请输入一个非负整数: 12345

0011000000111001

► 程序会在判断 **unsigned** 型的位数之后进行显示。因此，如果 **unsigned** 型为 32 位，就会显示 32 位。

按位操作的逻辑运算

本节学习按位（以位为单位）操作的“逻辑运算”。首先来看一下图 7-12 所示的逻辑运算真值表。

逻辑与 (AND)			逻辑或 (OR)			逻辑异或 (XOR)			逻辑非 (NOT)	
	0	1		0	1		0	1	0	1
0	0	0	0	0	1	0	0	1	1	0
1	0	1	1	1	1	1	1	0		

图 7-12 逻辑运算真值表

逻辑与在两个条件同时为真（1）的情况下，运算结果为真。**逻辑或**在任意一个条件为真的情况下，运算结果为真。**逻辑异或**在两个条件中只有一个条件为真的情况下，运算结果为真。**逻辑非**为取反，真的情况为假（0），假的情况为真。

假设整数 na 和 nb 都有 4 位，它们的值分别是二进制数 0101 和 0011。对 na 和 nb 的每一位计算逻辑与、逻辑或、逻辑异或以及对 na 的每一位求逻辑非的结果如下。

	逻辑与	逻辑或	逻辑异或	逻辑非
na	0101	0101	0101	0101
nb	0011	0011	0011	
	0001	0111	0110	1010

这些逻辑运算需要用到以下 4 个运算符：

按位与运算符（bitwise AND operator）： $\&$ 运算符

按位或运算符（bitwise inclusive OR operator）： $|$ 运算符

按位异或运算符（bitwise exclusive OR operator）： \wedge 运算符

\sim 运算符（ \sim operator） ※ 通常称为**按位求反运算符**

这些运算符的功能说明如表 7-2 所示。

表 7-2 按位运算符

按位与运算符	$a \& b$	按位计算 a 和 b 的逻辑与。
按位或运算符	$a b$	按位计算 a 和 b 的逻辑或。
按位异或运算符	$a \wedge b$	按位计算 a 和 b 的逻辑异或。
\sim 运算符	$\sim a$	计算 a 的反码（将每一位取反之后的值）。

► \sim 运算符是单目运算符的一种。

代码清单 7-6 所示程序的功能是将读取到的两个非负整数按位进行逻辑与和逻辑或等运算，并显示运算结果。

代码清单 7-6

```
/*
 * 显示unsigned型的逻辑与、逻辑或、逻辑异或和反码
 */

#include <stdio.h>

int count_bits(unsigned x)
{ /*---省略(参考代码清单7-5)---*/ }

int int_bits(void)
{ /*---省略(参考代码清单7-5)---*/ }

void print_bits(unsigned x)
{ /*---省略(参考代码清单7-5)---*/ }

int main(void)
{
    unsigned na, nb;

    puts("请输入两个非负整数。");
    printf("整数A:"); scanf("%u", &na);
    printf("整数B:"); scanf("%u", &nb);

    printf("\n A      ="); print_bits(na);
    printf("\n B      ="); print_bits(nb);
    printf("\n A & B ="); print_bits(na & nb);
    printf("\n A | B ="); print_bits(na | nb);
    printf("\n A ^ B ="); print_bits(na ^ nb);
    printf("\n ~ A  ="); print_bits(~na);
    printf("\n ~ B  ="); print_bits(~nb);
    putchar('\n');

    return (0);
}
```

运行结果

请输入两个非负整数。

整数A: 330

整数B: 440

```
A      = 0011000000111001
B      = 1101010000110001
A & B  = 0001000000110001
A | B  = 1111010000111001
A ^ B  = 1110010000001000
~ A    = 1100111111000110
~ B    = 0010101111001110
```

```
/* 逻辑与 */
/* 逻辑或 */
/* 逻辑异或 */
/* 反码 */
/* 反码 */
```

专题 7-2 逻辑运算符和按位逻辑运算符

现在学习的 &、|、~ 运算符的写法和功能都同 &&、||、! 运算符相似，所以要注意它们的区别。

逻辑运算包括逻辑与、逻辑或、逻辑异或、逻辑非、逻辑与非、逻辑或非等运算，运算结果只有“真”和“假”两种取值。

&、|、~ 运算符会根据 1 为真、0 为假的规则对操作数的各二进制位进行逻辑运算。

&&、||、! 运算符会根据非 0 为真、0 为假的规则对操作数的值进行逻辑运算。

我们通过比较表达式 5 & 4 和 5 && 4 的结果就能非常清楚地知道两者的区别了。

```
5 & 4    → 4          5 && 4    → 1
101 & 100 → 100      非 0 && 非 0 → 1
```

位移运算符

C 语言中可以自由方便地对构成整数的位进行左移或右移操作。**<< 运算符 (<< operator)** 和 **>> 运算符 (>> operator)** 的作用是求出左移或右移整数中的位之后的值。这两个运算符统称为**位移运算符 (bitwise shift operator)** (见表 7-3)。

在代码清单 7-7 所示的程序中对读取到的无符号整数进行了位移操作。

表 7-3 位移运算符

<< 运算符	$a \ll b$ 将 a 左移 b 位。右面空出的位用 0 填充。
>> 运算符	$a \gg b$ 将 a 右移 b 位。

左移

$nx \ll no$ 会将 nx 左移 no 位，并将右面空出的位用 0 填充 (图 7-13)。如果 nx 是无符号整数型，那么运算结果就相当于 nx 乘以 2^{no} 。

► 二进制数的每一位都是 2 的指数幂，所以左移 1 位后值变为原来的 2 倍 (十进制数左移 1 位后，值变为原来的 10 倍)。

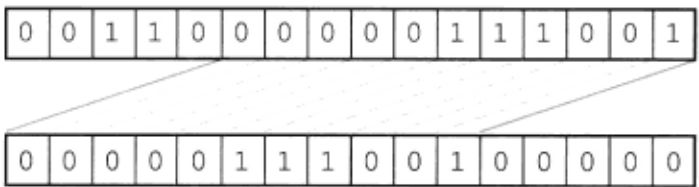


图 7-13 左移示例

右移

$nx \gg no$ 会将 nx 右移 no 位。如果 nx 是无符号整数型或是有符号整数型的非负数，那么运算结果就相当于 nx 除以 2^{no} 所得的商的整数部分值。(图 7-14)

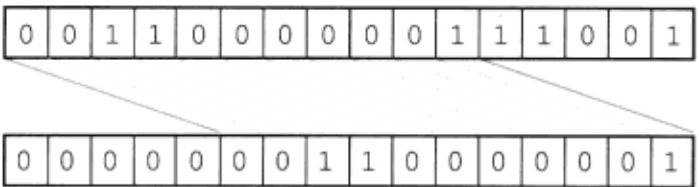


图 7-14 右移示例

当 nx 是有符号整数型的负数时，位移运算的结果因编译器而异。在许多编译器中，会执行**逻辑位移 (logical shift)** 或**算术位移 (arithmetic shift)** (参见专题 7-2)。

无论采用哪种方法都会降低程序的可移植性，所以我们要记住不要对负数作位移。

► 执行左移和右移操作时要注意不要对负数作位移，也不要作超过位数的位移。

代码清单 7-7

```

/*
 显示对 unsigned 型作左移和右移后的值
*/

#include <stdio.h>

int count_bits(unsigned x)
{ /*--- 省略 (参考代码清单 7-5) ---*/ }

int int_bits(void)
{ /*--- 省略 (参考代码清单 7-5) ---*/ }

void print_bits(unsigned x)
{ /*--- 省略 (参考代码清单 7-5) ---*/ }

int main(void)
{
    unsigned nx, no;

    printf("请输入一个非负整数:");
    scanf("%u", &nx);
    printf("位移位数:");
    scanf("%u", &no);

    printf("\n 整数      = ");    print_bits(nx);
    printf("\n 左移后的值 = ");    print_bits(nx << no);
    printf("\n 右移后的值 = ");    print_bits(nx >> no);
    putchar('\n');

    return (0);
}

```

运行结果

请输入一个非负整数: 12345
 位移位数: 5
 整数 = 0011000000111001
 左移后的值 = 0000011100100000
 右移后的值 = 0000000110000001

专题 7-3 逻辑位移和算术位移

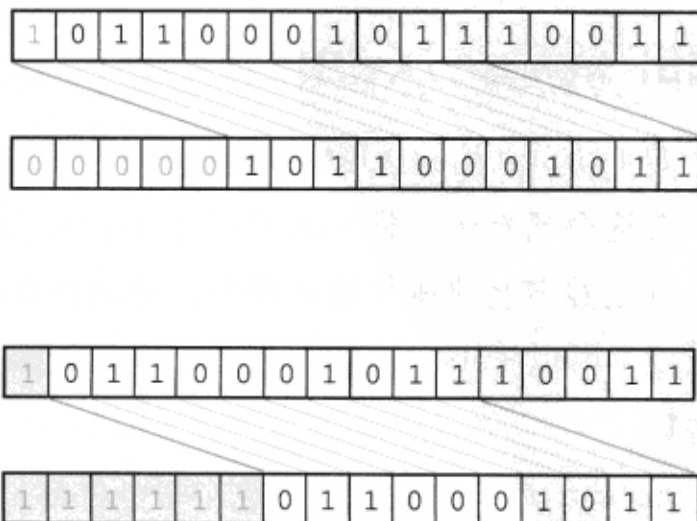
逻辑位移

如右图所示，逻辑位移不考虑符号位，所有二进制位都作位移。

符号位由 1 变为 0，位移的结果由负数变为正数。

算术位移

如右图所示，算术位移会保留最高位的符号位，只有其他位会作位移。除了保留符号以外，还维持“左移 1 位，值变为原来的 2 倍；右移 1 位，值变为原来的 1/2”的规则。



位数的计算

在学完位运算符之后，我们回到代码清单 7-5 所示的程序。

程序开头的 `count_bits` 函数的功能是计算形参 `x` 所接收到的无符号整数中有多少个值为 1 的二进制位，并返回其个数。

如果 `x` 的最低位是 1，那么 `x` 和 1U（只有最低位是 1，其他位都是 0 的无符号整数）的逻辑与运算结果就是 1。程序中的 `>>=` 运算符是一种复合赋值运算符。它和 `x=x>>1`；的作用是一样的。

```
int count_bits(unsigned x)
{
    int count = 0;
    while (x) {
        if (x & 1U) count++;
        x >>= 1;
    }
    return (count);
}
```

图 7-15 展示了如何使用 `while` 语句计算整数 10 中有多少个值为 1 的二进制位。相信看过图示之后就能理解程序的流程了。



图 7-15 位数计算示例

求出 unsigned 型的位数

`int_bits` 函数会返回 `unsigned` 型的位数。

在这个函数中，将 `~0U` 传给了 `count_bits` 函数。由于 `~` 运算符会将操作数中所有二进制位 0 和 1 的值逐位取反，所以 `~0U` 就变成了所有二进制位都是 1 的无符号整数。

```
int int_bits(void)
{
    return (count_bits(~0U));
}
```

`count_bit` 函数会返回这个每一位都是 1 的整数的位数，所以最终 `int_bits` 函数会返回 `unsigned` 型的位数。

显示位的内容

这个程序的主角是 `print_bits` 函数。该函数会显示 `unsigned` 型整数的各个二进制位的内容。

请试着解读这段代码。

```
void print_bits(unsigned x)
{
    int i;
    for (i = int_bits() - 1; i >= 0; i--)
        putchar(((x >> i) & 1U) ? '1': '0');
}
```

● 练习 7-1

编写程序确认：

无符号整数位左移后的值等于乘以 2 的指数幂后的值。

无符号整数位右移后的值等于除以 2 的指数幂后的值。

● 练习 7-2

编写 `rrotate` 函数，返回无符号整数 `x` 右移 `n` 位后的值。

```
unsigned rrotate(unsigned x, int n) { /* ... */ }
```

编写 `lrotate` 函数，返回无符号整数 `x` 左移 `n` 位后的值。

```
unsigned lrotate(unsigned x, int n) { /* ... */ }
```

● 练习 7-3

编写 `set` 函数，返回将无符号整数 `x` 的第 `pos` 位设为 1 后的值。

```
unsigned set( unsigned x, int pos) { /* ... */ }
```

编写 `reset` 函数，返回将无符号整数 `x` 的第 `pos` 位设为 0 后的值。

```
unsigned reset( unsigned x, int pos) { /* ... */ }
```

编写 `inverse` 函数，返回将无符号整数 `x` 的第 `pos` 位取反后的值。

```
unsigned inverse(unsigned x, int pos) { /* ... */ }
```

● 练习 7-4

编写 `set_n` 函数，返回将无符号整数 `x` 的第 `pos` 位开始的 `n` 位设为 1 后的值。

```
unsigned set_n( unsigned x, int pos, int n) { /* ... */ }
```

编写 `reset_n` 函数，返回将无符号整数 `x` 的第 `pos` 位开始的 `n` 位设为 0 后的值。

```
unsigned reset_n( unsigned x, int pos, int n) { /* ... */ }
```

编写 `inverse_n` 函数，返回将无符号整数 `x` 的第 `pos` 位开始的 `n` 位取反后的值。

```
unsigned inverse_n(unsigned x, int pos, int n) { /* ... */ }
```

整数的显示

在第 1 章开头的程序说明中，提到了下述内容。

printf 函数的第一个实参 **"%d"** 指定了输出格式，它告诉程序：
以十进制数的形式显示后面的实参。

printf 函数能够以八进制数和十六进制数进行输出。输出八进制数时使用 **"%o"**，输出十六进制数时使用 **"%X"** 或 **"%x"**。

► 十六进制的 A~F 在 **"%x"** 时显示为小写字母，在 **"%X"** 时显示为大写字母。

代码清单 7-8 所示的程序分别以十进制、二进制、八进制和十六进制形式显示了 0 ~ 65535 的整数。

代码清单 7-8

```
/*
 以十进制、二进制、八进制和十六进制形式显示 0 ~ 65535
*/

#include <stdio.h>

int count_bits(unsigned x)
{ /*--- 省略（参考代码清单 7-5） ---*/ }

int int_bits(void)
{ /*--- 省略（参考代码清单 7-5） ---*/ }

/*--- 显示 unsigned 型整数 x 的后 n 位 ---*/
void print_nbits(unsigned x, unsigned n)
{
    int i = int_bits();
    i = (n < i) ? n - 1 : i - 1;
    for ( ; i >= 0; i--)
        putchar((x >> i) & 1U ? '1' : '0');
}

int main(void)
{
    unsigned i;

    for (i = 0; i <= 65535U; i++) {
        printf("%5u ", i);
        print_nbits(i, 16);
        printf(" %06o %04X\n", i, i);
    }

    return (0);
}
```

运行结果			
0	0000000000000000	0	0
1	0000000000000001	1	1
2	0000000000000010	2	2
3	0000000000000011	3	3
(中略)			
65532	1111111111111100	177774	FFFC
65533	1111111111111101	177775	FFFD
65534	1111111111111110	177776	FFFE
65535	1111111111111111	177777	FFFF

数据溢出和异常

如果在 `int` 型可表示的数值范围为 $-32768 \sim 32767$ 的编译器中进行下述运算，结果会如何呢？

```
int x = 30000;
int y = 20000;
int z;
z = x + y;
```

`x` 和 `y` 中保存的值可以用 `int` 型来表示，这点毋庸置疑。但是赋给 `z` 的 50000 却超出了 `int` 型的表示范围。

像这样，因**数据溢出**（overflow）（溢位）使运算结果超出可表示的数值范围或违反数学定义（除以 0 等）时会发生**异常**（exception）。

发生异常时程序如何运行是由编译器决定的。

*

不过，无符号整数型的运算不会发生数据溢出。例如，我们在 `unsigned` 型数值范围为 $0 \sim 65535$ 的编译器中执行以下运算，看看结果如何。

```
unsigned x = 60000;
unsigned y = 7000;
unsigned z;
z = x + y;
```

`x` 与 `y` 的和在数学定义中应为 67000。但是经过运算之后就超过了无符号整数的表示范围。运算结果为除以 1 与该数据类型可表示的最大值的和之后所得的余数。所以将 67000 除以 65536 的余数 1464 赋给了 `z`。

同理，65535 加 1 的结果为 0。即循环使用 $0 \sim$ 最大值。

● 练习 7-5

编写程序确认对无符号整数执行算术运算不会发生数据溢出。

7-3 浮点型

浮点型

浮点型可以表示具有小数部分的实数，它有以下 3 种类型。

float double long double

这 3 种类型可表示的数值范围大于或等于各自左边的数据类型。

► 和整型的 `<limits.h>` 相同，浮点型的特性是在 `<float.h>` 头文件中定义的。

整型和字符型可以表示“一定的数值范围”的连续整数。而浮点数有所不同，它的表示范围是由长度和精度共同决定的。例如“长度为 12 位数字，精度为 6 位有效数字”，下面以具体数值为例进行思考。

12345678901 ... (a)

这个数值有 11 位，长度在表示范围之内。但它在精度为 6 位时无法正确表示，所以将第 7 位四舍五入，得到：

12345700000 ... (b)

可用科学计数法将 (b) 表示为：

0.123457×10^{11}

其中 0.123457 称为**尾数**，11 称为**指数**。尾数的小数部分位数决定了“精度”，指数的位数决定了“长度”。

到目前为止我们都以十进制数为例进行思考，但实际上尾数部分和指数部分都是用二进制数表示的（因此，在诸如长度或精度为“6 位”的情况下，并不能用整数正确无误地表示）。

浮点型的内部是由符号、指数和尾数三部分构成的(见图 7-16)。数据类型的表示范围越大，指数部分和尾数部分所占的位数就越多。由于各个编译器的具体表示方法有着很大的差别，使用时请查阅自己所用的编译器参考手册。

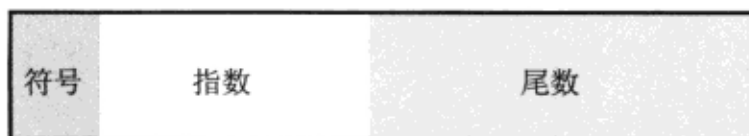


图 7-16 浮点数表示示例

浮点型常量

在浮点型常量中，可以加上浮点型后缀（floating suffix）。后缀 `f` 或 `F` 表示 **float** 型，后缀 `l` 或 `L` 表示 **long double** 型。例如：

```
80.0 ... double 型
80.0F ... float 型
80.0L ... long double 型
```

此外，还可以用科学计数法表示，如下面那样为数值加上指数等。

```
80.0E-5      /* 80.0 × 10-5 */
```

甚至可以像下面那样省略整数部分或小数部分。

```
.5           /* 0.5 */
10.          /* 10.0 */
```

但是不能将所有部分都省略。相关规则请参见图 7-17 中的语法图。

► 假如省略了小数点和小数部分，就必须给出整数部分。

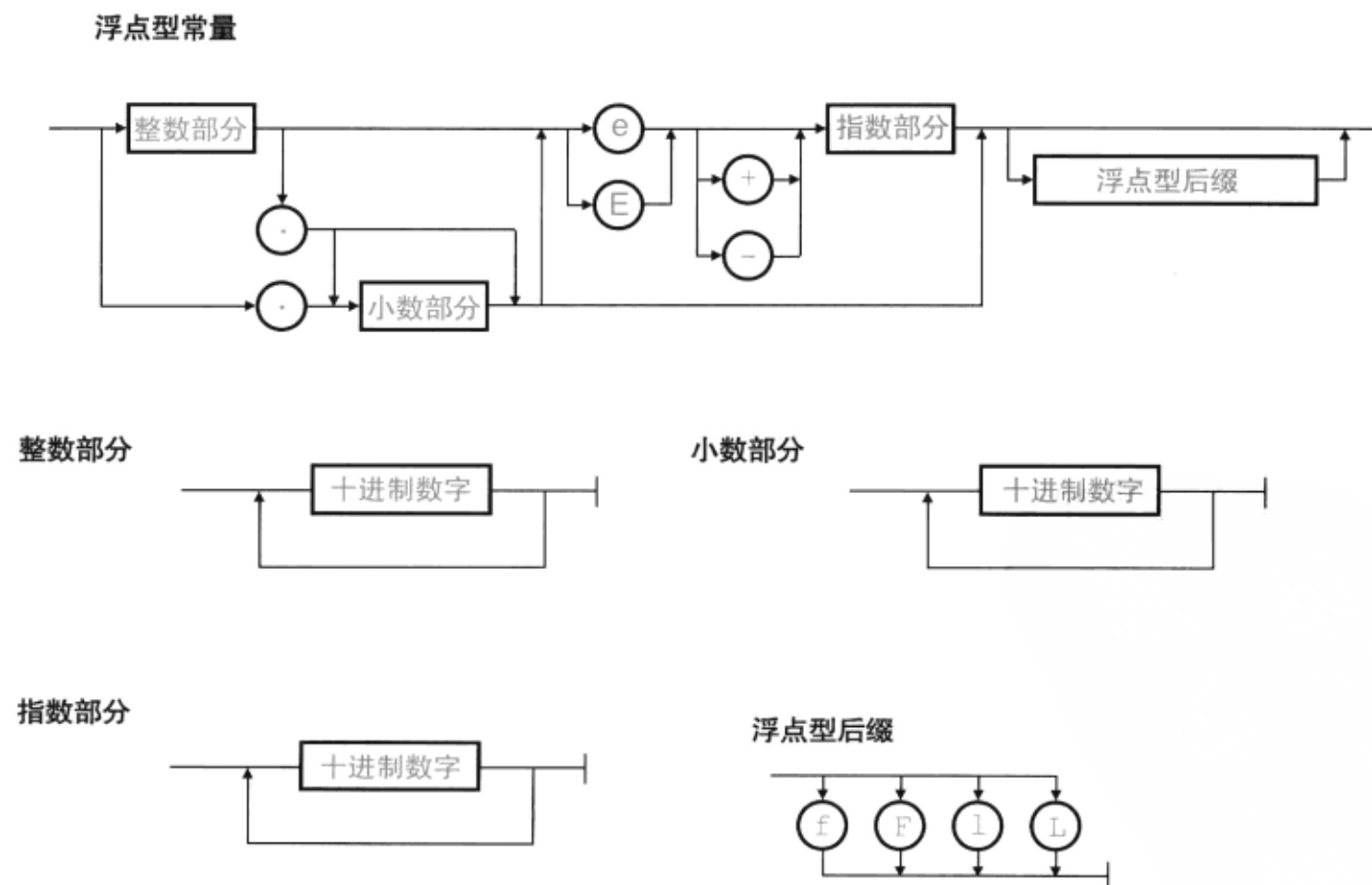


图 7-17 浮点型常量的结构图

循环的控制

请看代码清单 7-9 所示的程序。该程序显示了 **float** 型变量 x 以 0.01 为单位从 0.0 递增至 1.0 的每一步的结果（表示 **float** 型数值的规格转换为 **%f**）。

代码清单 7-9

```
/*
以 0.01 为单位从 0.0 递增至 1.0 的循环
*/
#include <stdio.h>

int main(void)
{
    float x;

    for (x = 0.0; x <= 1.0; x += 0.01)
        printf("x = %f\n", x);

    return (0);
}
```

运行结果

```
x = 0.000000
x = 0.010000
x = 0.020000
x = 0.030000
(中略)
x = 0.989999
x = 0.999999
```

► 运行结果或因编译器而异。

从运行结果可以发现，我们原以为结果应该正好是 1.0，可实际上却是 0.999999。计算机不能保证其内部转换为二进制的浮点数的每一位都不发生数据丢失。

如果 **for** 语句的控制表达式的判断为：

$x \neq 1.0$

那么这个 **for** 语句会跳过 1.0 继续循环下去。

代码清单 7-10

```
/*
以 0.01 为单位从 0.0 递增至 1.0 的循环（用整数控制）
*/
#include <stdio.h>

int main(void)
{
    int i;
    float x;

    for (i = 0; i <= 100; i++) {
        x = i / 100.0;
        printf("x = %f\n", x);
    }

    return (0);
}
```

运行结果

```
x = 0.000000
x = 0.010000
x = 0.020000
x = 0.030000
(中略)
x = 0.990000
x = 1.000000
```

► 运行结果或因编译器而异。

■ 注意 ■

循环判断基准所使用的变量应为整数而不要用浮点数。

用整数重新修改过的程序见代码清单 7-10。当然，这种写法也不一定能完全精确无误地表示实数数值，但至少不会像代码清单 7-9 那样积累误差。

<math.h> 头文件

C 语言提供了基本的数学函数来支持科学计算。<math.h> 头文件中包含了这些函数的声明。如代码清单 7-11 所示的程序，使用了求平方根的 **sqrt** 函数来计算两点间的距离。

sqrt

头文件 **#include <math.h>**

原型 **double sqrt(double x)**

说明 计算 **x** 的平方根（实参为负数时会发生定义域错误）。

返回值 返回计算后的平方根。

代码清单 7-11

```
/*
  求出两点间的距离
*/

#include <math.h>
#include <stdio.h>

/*--- 求出点 (x1,y1) 和点 (x2,y2) 之间的距离 ---*/
double dist(double x1, double y1, double x2, double y2)
{
    return (sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)));
}

int main(void)
{
    double x1, y1;          /* 点 1 */
    double x2, y2;          /* 点 2 */

    printf("《点 1》X坐标: ");    scanf("%lf", &x1);
    printf("          Y坐标: ");    scanf("%lf", &y1);
    printf("《点 2》X坐标: ");    scanf("%lf", &x2);
    printf("          Y坐标: ");    scanf("%lf", &y2);

    printf("2 点之间的距离为 %f。 \n", dist(x1, y1, x2, y2));

    return (0);
}
```

运行结果

《点 1》X 坐标: 1.5

Y 坐标: 2.0

《点 2》X 坐标: 3.7

Y 坐标: 4.2

2 点之间的距离为 3.111270。

7-4 运 算

运算符一览

至此我们已经学习了许多运算符。表 7-4 罗列了 C 语言中所有的运算符。

优先级

运算符一览表中，运算符越靠上，**优先级**（precedence）越高。例如，进行乘除法运算的 \star 和 $/$ 比进行加减法运算的 $+$ 和 $-$ 优先级高，这与我们实际生活中使用的数学规则是一样的。

$$a + b \star c$$

会被解释为 $a + (b \star c)$ ，而不是 $(a + b) \star c$ 。虽然 $+$ 写在前面，但还是先进行 \star 的运算。

结合性

这里有必要对**结合性**（associativity）作一下说明。假如用 \bigcirc 表示需要两个操作数的双目运算符，那么对于表达式 $a \bigcirc b \bigcirc c$

左结合运算符会将表达式解释为：

$$(a \bigcirc b) \bigcirc c \quad \text{左结合性}$$

右结合运算符会将表达式解释为：

$$a \bigcirc (b \bigcirc c) \quad \text{右结合性}$$

也就是说，遇到优先级相同的运算符时，结合性指明了表达式应从左往右运算还是从右往左运算。

例如，执行减法计算的双目运算符 $-$ 是左结合性的，所以

$$5 - 3 - 1 \rightarrow (5 - 3) - 1 \quad /* \text{左结合性} */$$

如果为右结合性，就会解释为 $5 - (3 - 1)$ ，答案就不正确了。执行赋值操作的简单赋值运算符 $=$ 是右结合性的，所以解释如下：

$$a = b = 1 \rightarrow a = (b = 1) \quad /* \text{右结合性} */$$

■ 表 7-4 运算符一览表

优先级	运算符	形式	名称 (通称)	结合性	参考
1	()	$x(y)$	函数调用运算符	左	115
	[]	$x[y]$	下标运算符	左	89
	.	$x.y$. 运算符 (句点运算符)	左	274
	->	$x->y$	-> 运算符 (箭头运算符)	左	276
	++	$x++$	后置递增运算符	左	67
	--	$y--$	后置递减运算符	左	67
2	++	$++x$	前置递增运算符	右	73
	--	$--y$	前置递减运算符	右	73
	sizeof	sizeof x	sizeof 运算符	右	180
	&	$\&x$	单目运算符 & (取址运算符)	右	228
	*	$*x$	单目运算符 * (指针运算符)	右	231
	+	$+x$	单目运算符 +	右	22
	-	$-x$	单目运算符 -	右	22
	~	$\sim x$	~ 运算符 (按位求补运算符)	右	164
3	!	$!x$	逻辑非运算符	右	61
	()	$(x)y$	类型转换运算符	右	31
4	*	$x*y$	双目运算符 *	左	19
	/	x/y	/ 运算符	左	19
	%	$x\%y$	% 运算符	左	19
5	+	$x+y$	双目运算符 +	左	19
	-	$x-y$	双目运算符 -	左	19
6	<<	$x<<y$	<< 运算符	左	166
	>>	$x>>y$	>> 运算符	左	166
7	<	$x<y$	< 运算符	左	44
	<=	$x<=y$	<= 运算符	左	44
	>	$x>y$	> 运算符	左	44
	>=	$x>=y$	>= 运算符	左	44
8	==	$x==y$	== 运算符	左	42
	!=	$x!=y$!= 运算符	左	42
9	&	$x\&y$	按位与运算符	左	164
10	^	$x\^y$	按位异或运算符	左	164
11		$x y$	按位或运算符	左	164
12	&&	$x\&\&y$	逻辑与运算符	左	53
13		$x y$	逻辑或运算符	左	53
14	? :	$x?y:z$	条件运算符	右	48
15	=	$x=y$	基本赋值运算符	右	23
	+= -= *= /= %= <<= >>= &= ^= =		复合赋值运算符	右	66
16	,	x,y	逗号运算符	左	110

数据类型转换

我们在第2章中简单地学习了数据类型转换。本节将说明详细规则，请在需要时作为参考（其中有些语句未展开说明）。

■ 整型提升

在可以使用 **int** 型或 **unsigned int** 型的表达式中，也可以使用有符号或无符号类型的 **char**、**short int**、**int** 位域，还可以使用枚举对象。

无论哪种情况，如果用 **int** 型可以表示出原数据类型的所有数值，就将值转换为 **int** 型，否则转换为 **unsigned int** 型。

► 整型提升不会改变符号和数值。**char** 型是否作为有符号类型来处理，由编译器而定。

■ 有符号整型和无符号整型

整数类数据类型之间互相转换时，若原数值能用转换后的数据类型表示，则数值不会发生变化。

将有符号整数转换为位数相同或位数更多的无符号整数时，如果该有符号整数不为负数，则数值不会发生变化。

否则，若无符号整数的位数较大，则先将有符号整数提升为与无符号整数长度相同的有符号整数。然后再与无符号整数类型可表示的最大数加1后的值相加，将有符号整数转换为无符号整数。

将整数类数据类型转换为位数更少的无符号整数时，除以比位数较少的数据类型可表示的最大无符号数大1的数，所得的非负余数就是转换后的值。

将整数类数据类型转换为位数更少的有符号整数时，以及将无符号整数转换为位数相同的有符号整数时，如果不能正确表示转换后的值，则此时的操作由编译器而定。

■ 浮点型和整数类数据类型

将浮点型的值转换为整数类数据类型时，会截断小数部分。整数部分的值不能用整数类数据类型表示时的操作未定义。

将整数类数据类型的值转换为浮点型时，如果数据类型转换后的结果在数值范围内不能正确表示，那么会根据编译器定义的方法取大于或小于原值的最接近的近似值作为转换结果。

■ 浮点型

float 型提升为 **double** 型或 **long double** 型时，以及 **double** 型提升为 **long double** 型时，值不会发生变化。

double 型转换为 **float** 型时，以及 **long double** 型转换为 **float** 型时，会根据编译器定义的方法取大于或小于原值的最接近的近似值作为转换结果。

■ 普通算术类型转换

许多具有算术类型操作数的双目运算符都会执行操作数的数据类型转换，并用同样的方法决定转换结果的数据类型。数据类型转换的目的是确定通用数据类型，该数据类型亦是转换结果的数据类型。这一过程称为**普通算术类型转换**（usual arithmetic conversion）。普通算术类型转换的规则如下：

- (a) 若有一个操作数为 **long double** 型，则将另一个操作数转换为 **long double** 型。
- (b) 若有一个操作数为 **double** 型，则将另一个操作数转换为 **double** 型。
- (c) 若有一个操作数为 **float** 型，则将另一个操作数转换为 **float** 型。
- (d) 若均不符合以上情况，则根据以下规则对两个操作数进行整型提升。
 - (1) 若有一个操作数为 **unsigned long** 型，则将另一个操作数转换为 **unsigned float** 型。
 - (2) 在一个操作数为 **long** 型，另一个操作数为 **unsigned** 型的情况下，如果 **long** 型能表示 **unsigned** 型的所有值，则将 **unsigned** 型的操作数转换为 **long** 型。如果 **long** 型不能表示 **unsigned** 型的所有值，则将两个操作数都转换为 **unsigned long** 型。
 - (3) 若有一个操作数为 **long** 型，则将另一个操作数转换为 **long** 型。
 - (4) 若有一个操作数为 **unsigned** 型，则将另一个操作数转换为 **unsigned** 型。
 - (5) 将两个操作数都转换为 **int** 型。

浮点型操作数的值以及浮点型表达式的结果值可以超出数据类型所要求的精度和范围进行显示。但是结果的数据类型不会发生变化。

► 有关数据类型转换的内容，将在另一本书《明解 C 语言：实践篇》中进行详细说明。

sizeof 运算符

我们在前面学习了判断数据类型长度的 **sizeof** 运算符（表 7-5）。

sizeof （数据类型名称）

该运算符的用法不仅仅是上面这种形式，还可以像下面这样使用：

sizeof 表达式

要判断数据类型长度时使用前者，要判断变量或表达式的长度时使用后者。

▶ 后者在形式上不要求将表达式用（）括起来，但为了保持上下文统一，建议加上（）。

表 7-5 sizeof 运算符

sizeof 运算符	sizeof a	求出 a（对象、常量、数据类型名称等）的长度。
-------------------	-----------------	-------------------------

代码清单 7-12 所示的程序显示了 **int** 型和 **double** 型变量的长度。

代码清单 7-12

```
/*
 * 显示数据类型和变量的长度
 */
#include <stdio.h>

int main(void)
{
    int      na, nb;
    double   dx, dy;

    printf("sizeof(int)      = %u\n", (unsigned)sizeof(int));
    printf("sizeof(double)   = %u\n", (unsigned)sizeof(double));

    printf("sizeof(na)       = %u\n", (unsigned)sizeof(na));
    printf("sizeof(dx)       = %u\n", (unsigned)sizeof(dx));

    printf("sizeof(na + nb) = %u\n", (unsigned)sizeof(na + nb));
    printf("sizeof(na + dy) = %u\n", (unsigned)sizeof(na + dy));
    printf("sizeof(dx + dy) = %u\n", (unsigned)sizeof(dx + dy));

    return (0);
}
```

运行结果

```
sizeof(int)      = 2
sizeof(double)   = 8
sizeof(na)       = 2
sizeof(dx)       = 8
sizeof(na + nb) = 2
sizeof(na + dy) = 8
sizeof(dx + dy) = 8
```

▶ 运行结果因编译器而异。

sizeof 运算符和数组

将 **sizeof** 运算符用于数组就会得到数组的长度。因此，在 **sizeof(int)** 为 2 的编译器中，对于用

```
int vc[5];
```

声明的数组，可以使用

```
sizeof(vc)
```

求出它的长度，结果为 10。

掌握了这点之后，就可以使用下述表达式求出数组的元素个数了。

```
sizeof(vc) / sizeof(vc[0])
```

当然，无论 *vc* 的元素数据类型是什么，这个表达式都能正确地求出元素个数。可以使用代码清单 7-13 所示的程序进行确认。

代码清单 7-13

```
/*
  求出数组的元素个数
*/

#include <stdio.h>

int main(void)
{
    int      vi[10];
    double   vd[25];

    printf("数组 vi 的元素个数 = %u\n", (unsigned)(sizeof(vi) / sizeof(vi[0])));
    printf("数组 vd 的元素个数 = %u\n", (unsigned)(sizeof(vd) / sizeof(vd[0])));

    return (0);
}
```

运行结果

数组 vi 的元素个数 = 10
数组 vd 的元素个数 = 25



第 8 章

动手编写各种程序吧

本章我们会引出几个问题，并以此来学习函数式宏、枚举类型、递归以及输入输出等。



8-1 函数式宏

函数和数据类型

我们来编写这样一个程序，它能计算出所读取数值的平方，并将结果显示出来。

当然，“数”有各种类型。我们这次先来编写适用于 **int** 型和 **double** 型的函数。请看代码清单 8-1。

代码清单 8-1

```
/*
  整数的平方和浮点数的平方（函数）
*/

#include <stdio.h>

/*--- int 型整数的平方值 ---*/
int sqr_int(int x)
{
    return (x * x);
}

/*--- double 型浮点数的平方值 ---*/
double sqr_double(double x)
{
    return (x * x);
}

int main(void)
{
    int        nx;
    double     dx;

    printf("请输入一个整数: ");
    scanf("%d", &nx);
    printf("该数的平方是 %d.\n", sqr_int(nx));

    printf("请输入一个实数: ");
    scanf("%lf", &dx);
    printf("该数的平方是 %f.\n", sqr_double(dx));

    return (0);
}
```

运行结果

请输入一个整数: 3
该数的平方是 9。
请输入一个实数: 4.25
该数的平方是 18.062500。

上一章我们学习了很多数据类型。假如现在还需计算 **long** 型数据的平方，那就得再编写一个名为 `sqr_long` 的函数了吧。

不过，若是接二连三地写出这种功能相近、名称又相似的函数，程序中就会充斥着这些似是而非的函数。

函数式宏

函数式宏 (function-like macro) 较之对象式宏可以进行更复杂的代换。代码清单 8-2 是使用函数式宏改写后的程序。

代码清单 8-2

```
/*
 * 整数的平方和浮点数的平方 (函数式宏)
 */

#include <stdio.h>

#define sqr(x) ((x) * (x))          /* 计算 x 的平方 */

int main(void)
{
    int      nx;
    double   dx;

    printf("请输入一个整数: ");
    scanf("%d", &nx);
    printf("该数的平方是 %d.\n", sqr(nx));

    printf("请输入一个实数: ");
    scanf("%lf", &dx);
    printf("该数的平方是 %f.\n", sqr(dx));

    return (0);
}
```

运行结果

请输入一个整数: 3
该数的平方是 9。
请输入一个实数: 4.25
该数的平方是 18.062500。

#define 命令给出的指示具体如下。

下文若出现 `sqr(○)` 形式的表达式, 就将其展开为
`((○) * (○))`

因此两处调用 **printf** 函数的部分, 可以像图 8-1 那样展开并进行编译和运行。

函数式宏和函数完全是两码事, 不过它看上去很像函数, 而且在使用上并不依赖于具体数据类型。

```
printf("该数的平方是%d.\n", sqr(nx));
                        ↓ 展开
printf("该数的平方是%d.\n", ((nx) * (nx)));

printf("该数的平方是%f.\n", sqr(dx));
                        ↓ 展开
printf("该数的平方是%f.\n", ((dx) * (dx)));
```

图 8-1 函数式宏的展开

函数和函数式宏

函数式宏的调用看上去和函数调用相同，那么这两者有何区别呢？主要有以下几个方面。

■ 函数式宏 `sqr` 是在编译时展开并填入程序的，因此可以应用于 **int** 型、**double** 型和 **long** 型数据。即只要是能用双目运算符 `*` 进行乘法计算的数据类型，都能使用函数式宏。

而函数定义则需为每个形参都定义各自的数据类型，返回值的类型也只能为一种。就这一点而言，函数较为严格。

■ 函数为我们默默无闻地进行了一些复杂处理，如：

- 参数传递（将实参的值复制到形参）
- 函数调用和函数返回操作（程序流程的控制）
- 返回值的传递

而函数式宏所做的工作只是宏展开和填入程序，并不进行上述处理。

■ 根据以上特征，函数式宏或许能使程序的运行速度稍微提高一点，但是程序自身却有可能变得臃肿（如果宏展开后的表达式很复杂，那么在使用到它的所有地方都会填入这些复杂的表达式）。

■ 函数式宏在使用上必须小心谨慎。例如，`sqr(a++)` 展开后为：

`((a++) * (a++))`

每次展开，`a` 的值都会自增两次。

在不经意间表达式被执行了两次，导致程序出现预料之外的结果，我们称这种情况为副作用。

■ 注意 ■

在定义和使用函数式宏时，请仔细考虑是否会产生副作用。

专题 8-1 函数式宏和对象式宏

如果在宏名称 `sqr` 和紧邻其后的 “`(`” 之间插入空格，进行如下宏定义

```
#define sqr (x) ((x)*(x))
```

就会被编译器当作对象式宏，即程序中的 `sqr` 都会被代换为 `(x)((x)*(x))`。

我们在定义函数式宏时必须注意不要误将空格写入宏名称和 “`(`” 之间。

不带参数的函数式宏

函数式宏也可以像函数那样进行不带参数的定义，例如下面这个响铃的宏 `alert()`。

```
#define alert() (putchar('\a'))
```

请试着写出调用该函数式宏的程序。

► 以下是计算二值之和的函数式宏。

```
#define add(x, y) x + y
```

我们使用下述语句来调用这个宏。

```
z = add(a, b) * add(c, d);
```

宏展开后的表达式不尽如人意。

```
z = a + b * c + d;
```

保险起见，我们在宏定义时将每个参数以及整个表达式都用 `()` 括起来就不会出错了。

```
#define add(x, y) ((x) + (y))
```

这样表达式就能正确展开了。

```
z = ((a) + (b)) * ((c) + (d));
```

● 练习 8-1

请定义一个函数式宏 `diff(x, y)`，返回 `x`、`y` 二值之差。

● 练习 8-2

现定义如下函数式宏，其功能为返回 `x`、`y` 中的较大值。

```
#define max(x, y) (((x) > (y)) ? (x) : (y))
```

而下面两个使用了该宏的表达式的功能为计算 `a`、`b`、`c`、`d` 中的最大值。

```
max(max(a, b), max(c, d))
```

```
max(max(max(a, b), c), d)
```

请显示并观察它们是如何展开的。

● 练习 8-3

请定义一个函数式宏 `swap(type, a, b)` 以使 `type` 型的两值互换。

例如：假设 `int` 型变量 `x`、`y` 的值分别为 5、10，那么调用 `swap(int, x, y)` 后，`x`、`y` 中应分别保存 10、5。

函数式宏和逗号运算符

本节将介绍函数式宏使用方面的一个重要技巧。请看代码清单 8-3。

代码清单 8-3

```
/*
 响铃并显示的宏定义（误例）
*/

#include <stdio.h>

#define putsa(str)    { putchar('\a'); puts(str); }

int main(void)
{
    int na;

    printf("请输入一个整数:");
    scanf("%d", &na);

    if (na)
        putsa("这个数不是 0。");
    else
        putsa("这个数是 0。");

    return (0);
}
```

► 这个程序不能编译和运行。

函数式宏 `putsa` 的定义是在 `puts` 函数显示字符串时响铃。不过，这个程序在编译时会出错，不能运行。

我们将程序中 `if` 语句部分展开，一起来看一下。

```
if (na)
    { putchar('\a'); puts("这个数不是 0。"); };
else
    { putchar('\a'); puts("这个数是 0。"); };
```

将它改写成图 8-2 的样子，错误就浮出水面了。我们不难发现分号“;”是多余的。它使 `if` 语句结束于第一个复合语句 `{ }`，因此编译器会认为“没有 `if`，为何出现了 `else`”。

if 语句	<code>if (na) { putchar('\a'); puts("这个数不是 0。"); }</code>
空语句	<code>;</code>
?	<code>else</code>
复合语句	<code>{ putchar('\a'); puts("这个数是 0。"); }</code>
空语句	<code>;</code>

图 8-2 宏展开后的语法

当然，我们也不能删除宏定义中的“{ }”（否则会发生其他错误，可以亲自动手试一下）。

这下就到了逗号运算符大显身手的时候了。在下面的代码清单 8-4 中，我们使用逗号运算符改写了宏定义 `putsa`。

代码清单 8-4

```
/*
  响铃并显示的宏定义
*/

#include <stdio.h>

#define putsa(str)    ( putchar('\a') , puts(str) )

int main(void)
{
    /*--- 省略（同代码清单 8-3） ---*/
}
```

如此一来，刚才有问题的地方可以展开如下。

```
if (na)
    ( putchar('\a'), puts("这个数不是 0。") );
else
    ( putchar('\a'), puts("这个数是 0。") );
```

一般由逗号运算符连接的两个表达式 `op1, op2` 在语法上可以视为一个表达式（其实不仅限于逗号运算符，只要是由运算符连接的多个表达式就可以视为一个表达式）。因此，这部分内容可以用图 8-3 来解释。

在表达式末尾加上分号就构成语句，因此这在语法上是正确的。

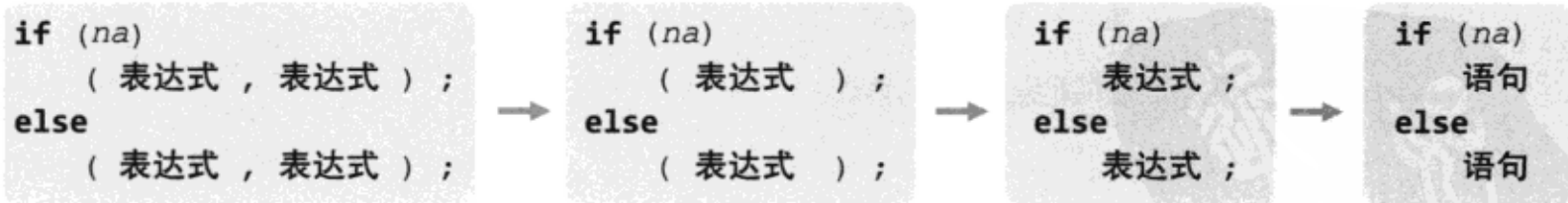


图 8-3 宏展开后的语法

■ 注意 ■

如果宏定义中要代换两个以上的表达式，则使用逗号运算符连接，使其在语法上构成一个表达式。

8-2 枚举类型

枚举类型

代码清单 8-5 的程序中给出了狗、猫、猴三个选项，做出选择后会显示所选动物的叫声。

代码清单 8-5

```

/*
 * 显示所选动物的叫声
 */

#include <stdio.h>

enum animal { Dog, Cat, Monkey, Invalid };

/*--- 狗叫 ---*/
void dog(void)
{
    puts("汪汪!!");
}

/*--- 猫叫 ---*/
void cat(void)
{
    puts("喵喵!!");
}

/*--- 猴叫 ---*/
void monkey(void)
{
    puts("唧唧!!");
}

/*--- 选择动物 ---*/
enum animal select(void)
{
    int tmp;

    do {
        printf("0...狗 1...猫 2...猴 3...结束:");
        scanf("%d", &tmp);
    } while (tmp < Dog || tmp > Invalid);
    return (tmp);
}

int main(void)
{
    enum animal selected;

    do {
        switch (selected = select()) {
            case Dog : dog(); break;
            case Cat : cat(); break;
            case Monkey : monkey(); break;
        }
    } while (selected != Invalid);
    return (0);
}

```

运行结果

```

0...狗 1...猫 2...猴 3...结束: 0
汪汪!!
0...狗 1...猫 2...猴 3...结束: 2
唧唧!!
0...狗 1...猫 2...猴 3...结束: 3

```


程序中蓝色底纹部分，是**枚举类型**（enumeration）的声明，它表示了所有可用值的集合。其中，*animal* 被称为**枚举名**（tag）。写在 { } 中的 Dog、Cat、Monkey、

数据类型名称	变量名
<code>enum animal</code>	<code>selected;</code>
<code>int</code>	<code>vx;</code>

Invalid 是**枚举常量**（enumeration constant）。这些枚举值从左往右依次赋值为整数 0、1、2、3。

`enum animal` 型，是表示这些值的集合的数据类型。

main 函数中的蓝字部分，是 `enum animal` 型变量 *selected* 的声明。

通过与下图比较，我们可以清楚地知道 `enum animal` 是数据类型名称，而 *selected* 是变量名。

通过这个声明，定义了变量 *selected* 的取值范围为 0、1、2、3。

select 函数的功能是显示动物选项并返回所选动物。注意观察 **do** 语句（该循环语句的作用是如果输入了 0、1、2、3 以外的值，就引导用户再次输入）的循环条件表达式。其中使用到了枚举常量 *Invalid*，它既不表示动物也无特别定义，乍一看完全没有意义。

► *Invalid* 意为“无效的”。

现在我们假设不使用这个枚举常量，会发生什么情况呢？无疑，循环条件表达式将改为：

```
tmp < Dog || tmp > Monkey + 1
```

如果此时增加第 4 种动物“海豹”，则枚举类型 *animal* 将随之改为：

```
enum animal { Dog, Cat, Monkey, Seal };
```

相应地，循环条件表达式也必须修改为：

```
tmp < Dog || tmp > Seal + 1
```

即每次增加动物时，都须要修改判断循环条件的循环条件表达式。

由此可见，看似无用的 *Invalid* 实际上大有用处呢！

► 枚举常量的数据类型是 **int** 型。因此在返回类型为 `enum animal` 型的 *select* 函数中，可以返回 **int** 型变量 *tmp* 的值。

为明确起见，也可以将返回值作如下强制转换。

```
return ((enum animal)tmp);
```



枚举常量

在上一节的程序中，我们从 0 开始按顺序为枚举常量定义了相应的整数值。实际上，这些值也能够根据需要任意设置，只要在枚举常量的名称后面写上赋值运算符“=”和值就行了。

例如，在以下定义中，

```
enum kyushu { Fukuoka, Saga = 5, Nagasaki };
```

Fukuoka 为 0，Saga 为 5，Nagasaki 为 6。即通过赋值运算符“=”赋值的枚举常量，其值即为给定值。没有给定值的枚举常量，其值为前一个枚举常量加 1。

另外，如果进行了如下声明

```
enum nama { Shibata, Washio = 0 };
```

那么 Shibata 和 Washio 都为 0。多个枚举常量允许具有相同的值。

还有，程序中的枚举名也是可以省略的。例如，可以进行如下声明：

```
enum { JANUARY = 1, FEBRUARY, /* (中略) */ , DECEMBER };
```

通过这种方式声明的枚举常量，可以在如下所示的 **switch** 语句中使用。

```
int month;
/* ... */
switch (month) {
    case JANUARY : /* ... */
    case FEBRUARY : /* ... */
    /* ... */
}
```

● 练习 8-4

请在程序中定义表示性别、季节等的枚举类型，并有效使用它们。

下面我们来归纳一下枚举类型的特征。

■ 使用宏定义实现上一页中表示月份的枚举类型，即

```
#define JANUARY 1
#define FEBRUARY 2
/* ... */
```

```
#define DECEMBER 12
```

这在程序中会占去 12 行，而且必须逐个定义它们的值。而使用枚举类型来声明，就可以非常简洁，只要 JANUARY 的值正确，其他值就不会有错（通过自动计算可得）。

■ 表示动物的 `enum animal` 型，只有定义过的值才有效，即有效值为 0、1、2、3。如果变量 `an` 是该类型，那么对于以下赋值语句

```
an = 5;
```

一些人性化的编译器将会发出警告信息，提示赋给 `an` 的是未定义的值。这样就更容易发现程序中的错误。然而，若 `an` 是 `int` 型变量，则不能进行这种检查。

■ 在有些验证程序行为的调试软件中，将枚举型变量的值显示为枚举值的名称，而不是整数值。变量 `selected` 的值显示为 `Dog`，而不是 0，这就更便于调试了。

■ 注意 ■

能用枚举类型表示的数据类型，应尽量用枚举类型来表示。

命名空间

枚举名和变量名分别属于不同的命名空间（name space），因此即便名称相同也能正确区分。打个比方，人名里的福冈和地名里的福冈，虽然名字相同但是性质不同，所以可以区分清楚。如果说“我去福冈”，马上就能知道指的是地名。

因此，我们也可以将 `enum animal` 型的变量命名为 `animal`，进行如下声明

```
enum animal animal;
```

► 有关命名空间更多的内容可以参考 12-1 节。

8-3 递 归

阶乘

递归，就是用自己来定义自己。

整数 n 的阶乘的定义即包含着这种递归思想，如：

$$(a) 0! = 1$$

$$(b) \text{ 若 } n > 0, \text{ 则 } n! = n \times (n - 1)!$$

例如，10 的阶乘 $10!$ ，根据 (b) 的定义可得：

$$10! = 10 \times 9!$$

而右边的 $9!$ 又可分解为

$$9! = 9 \times 8!$$

通常我们设求出 n 的阶乘的方法为 $\text{factorial}(n)$ ，那么 $10!$ 和 $9!$ 可表示如下：

$$10! = 10 \times \text{factorial}(9)$$

$$9! = 9 \times \text{factorial}(8)$$

即 $\text{factorial}(n)$ 为 n 和 $\text{factorial}(n-1)$ 的乘积。所以只要知道 $\text{factorial}(n-1)$ 的值，就能很容易地计算出结果了。

基于这个思路，我们写出了 factorial 函数，请看代码清单 8-6 的程序。

假如我们现在要计算 $3!$ 的值，只需调用 $\text{factorial}(3)$ 即可，此时的计算步骤如图 8-4 所示。

通过 $\text{factorial}(3)$ 函数调用语句启动 factorial 函数，将 3 传入形参 n ，函数就会返回：

$$3 * \text{factorial}(2)$$

但是要进行这个乘法计算，就必须先知道 $\text{factorial}(2)$ 的值，于是以 2 为参数再次调用 factorial 函数。而这次需要知道 $\text{factorial}(1)$ 的值，所以又一次调用 factorial 函数。

接下来注意参数 n 为 0 时的 factorial 函数。因为 n 的值为 0，于是函数直接返回 1，并回到调用该函数的 factorial 函数。

该 factorial 函数返回 $1 * \text{factorial}(0)$ 即返回 $1 * 1$ ，并回到 factorial 函数。随后再次返回 factorial 函数，最后返回结果 6。

代码清单 8-6

```

/*
  计算阶乘
*/

#include <stdio.h>

/*--- 返回阶乘的值 ---*/
int factorial(int n)
{
    if (n > 0)
        return (n * factorial(n - 1));
    else
        return (1);
}

int main(void)
{
    int num;

    printf("请输入一个整数: ");
    scanf("%d", &num);

    printf("该数阶乘为 %d. \n", factorial(num));

    return (0);
}

```

运行结果

请输入一个整数: 3
该数阶乘为 6。

► 如果输入数字大于 16，所得结果超出整数类型表示范围，结果会出错。如果要计算大于 16 的阶乘，必须使用 double 类型；如果还需要计算更大的数，还要改变类型。

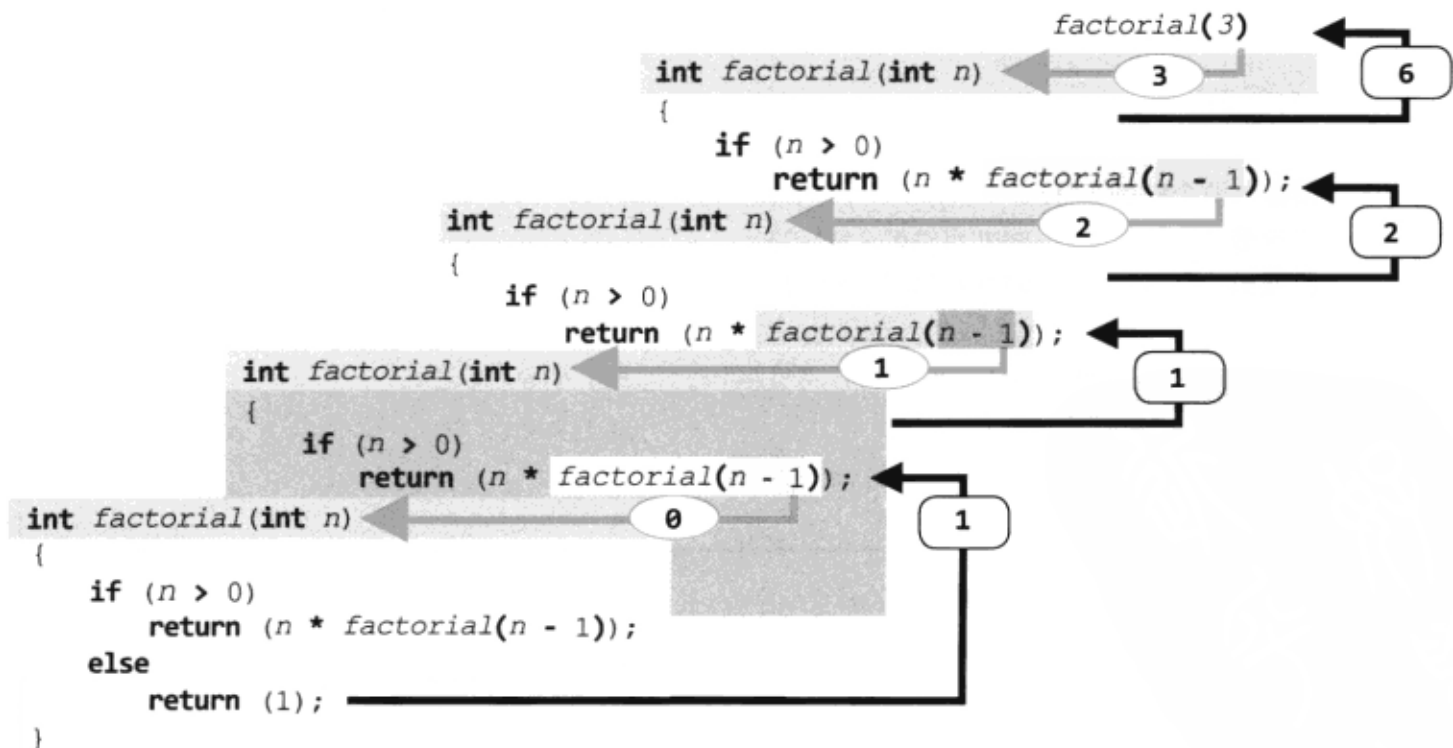


图 8-4 计算 3 的阶乘的步骤

像这样要进行某些计算或操作时，如果其实现方法正好是与自身相同的函数，就可以调用该函数。这种函数调用被称为递归函数调用（recursive function call）。

最大公约数

代码清单 8-7 所示程序的功能是求两个整数的最大公约数。

代码清单 8-7

```
/*
    求出最大公约数
*/

#include <stdio.h>

/*--- 返回 vx 和 vy 的最大公约数 (vx ≥ vy) ---*/
int gcdf(int vx, int vy)
{
    return (vy == 0 ? vx : gcdf(vy, vx % vy));
}

/*--- 求出 va 和 vb 的最大公约数 ---*/
int gcd(int va, int vb)
{
    return (va > vb ? gcdf(va, vb) : gcdf(vb, va));
}

int main(void)
{
    int n1, n2;

    puts("请输入两个整数。");
    printf("整数 1 : "); scanf("%d", &n1);
    printf("整数 2 : "); scanf("%d", &n2);

    printf("最大公约数是 %d。\\n", gcd(n1, n2));

    return (0);
}
```

运行结果

请输入两个整数。
 整数 1 : 8
 整数 2 : 22
 最大公约数是 2。

我们来思考一下求 8 和 22 最大公约数的方法。图 8-5 所示的是以这两个数为边长的长方形（长为 vx ，宽为 vy ）。

这个长方形可以分为两个以宽 vy （8）为边长的正方形和一个 8×6 的长方形。而从新的长方形（ $vx=8$ ， $vy=6$ ）中又可以分出以宽 vy （6）为边长的正方形。

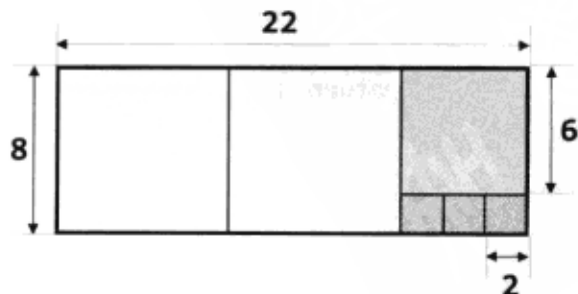


图 8-5 求 8 和 22 最大公约数的步骤

依此类推，反复进行“从剩下的长方形中分出正方形”的操作，直到全部都是正方形为止，此时的边长就是两数的最大公约数。

问题和递归

递归的强大之处在于可以用有限的描述来定义无限的现象。如果待处理的问题、函数或者数据结构已经具有递归定义，那么就可以使用递归算法。

*

有时递归调用会被说成调用“函数自身”，不过通过阶乘的说明图（图 8-4）也可发现实际上调用的并不是“函数自身”，而是另行调用了“与自身相同的函数”，我们必须弄清楚这个概念。因为要真是调用了“函数自身”，不就进入死循环了吗？

► 广为使用的编程语言 FORTRAN 和传统的 BASIC 语言不支持递归调用。

● 练习 8-5

不使用递归，定义如下函数，使其返回整数 n 的阶乘。

```
int fact(int n) { /* ... */ }
```

● 练习 8-6

编写如下函数，求出从 n 个不同整数中取出 r 个整数的组合数 C_n^r 。

```
int combination(int n, int r) { /* ... */ }
```

C_n^r 的定义如下：

$$C_n^r = C_{n-1}^{r-1} + C_{n-1}^r \quad (\text{且 } C_n^0 = C_n^n = 1, C_n^1 = n)$$

8-4 输入输出和字符

数字字符计数

请思考以下问题：

计算输入的数字字符数。

该程序如代码清单 8-8 所示。

getchar 函数

这里首次出现了 **getchar** 函数，它的功能是读入字符并将其返回。

getchar	
头文件	#include <stdio.h>
原 型	int getchar(void)
说 明	从标准输入流中读取下一个字符（若存在）。
返回值	返回读入的字符。读到文件末尾或发生错误时，返回 EOF。

读取过程中发生错误或输入结束时，就会返回 EOF 值。

EOF

EOF 意为 End Of File。例如在 <stdio.h> 头文件中有下述定义。

```
#define EOF -1
```

EOF 被定义为负值。

- 如果未将 <stdio.h> 头文件包含进来，那么 EOF 就没有定义，这时程序不能编译和运行。然而，在未包含 <stdio.h> 头文件的情况下，在程序中手动进行如下定义是不行的。

```
#define EOF -1
```

因为 EOF 在有些编译器中的定义为 -1，而在另一些编译器中的定义可能为 -2。因此在不同的编译器及运行环境中，这种程序的编译和运行结果的正确性是难以保证的。

这个程序的 **while** 语句的循环条件表达式是 1，因此会无条件地进行循环。不过当 **getchar** 函数的返回值为 EOF（输入结束）时，**break** 语句可以使程序跳出正常循环。

代码清单 8-8

```

/*
  计算标准输入流中出现的数字字符数
*/

#include <stdio.h>

int main(void)
{
    int i, ch;
    int cnt[10] = {0};          /* 数字字符的出现次数 */

    while (1) {                 /* 死循环 */
        ch = getchar();
        if (ch == EOF) break;

        switch (ch) {
            case '0' : cnt[0]++; break;
            case '1' : cnt[1]++; break;
            case '2' : cnt[2]++; break;
            case '3' : cnt[3]++; break;
            case '4' : cnt[4]++; break;
            case '5' : cnt[5]++; break;
            case '6' : cnt[6]++; break;
            case '7' : cnt[7]++; break;
            case '8' : cnt[8]++; break;
            case '9' : cnt[9]++; break;
        }

        puts("数字字符的出现次数");
        for (i = 0; i < 10; i++)
            printf("%d' : %d\n", i, cnt[i]);

        return (0);
    }
}

```

运行结果

3.1415926535897932846

Ctrl + Z

数字字符的出现次数

'0' : 0

'1' : 2

'2' : 2

'3' : 3

'4' : 2

'5' : 3

'6' : 2

'7' : 1

'8' : 2

'9' : 3

► 程序的注释、上一页的 `getchar` 函数说明中提到的“标准输入流”一词，以及运行结果中的 `Ctrl + Z` 操作的相关内容，会在本节最后详细介绍。

字符和数值

当 `getchar` 函数的返回值不为 `EOF` 时，即正确读入字符时，会进入 `switch` 语句。

'0'、'1'、'2'、'3'、'4'、'5'、'6'、'7'、'8'、'9'

`switch` 语句会对上面这十个数字的情况分别进行相应的处理，语句显得比较冗长。

数组 `cnt` 的作用是保存数字字符 '0' 到 '9' 的出现次数。由于其下标也是 0 到 9，因此只要将数字字符转换为对应的下标值就能更简单地实现同样功能。如将数字字符 '0' 转换为整数 0，将数字字符 '1' 转换为整数 1。

字符

上一章我们提到了字符型，但对“字符”的说明并没有深入展开。实际上 C 语言中的字符都作为非负整数值来处理。因此，每一个字符都有与之对应的编码（即整数值）。

注意

C 语言中的“字符”就是该字符的字符编码（即整数值）。

但是，即使是同一个字符，在不同的程序运行环境中编码也会有所不同。具体要看程序运行环境所用的字符编码。

日本大部分电脑所使用的字符编码为表 8-1 所示的“JIS 码”。我们就以它为例子进行说明。

首先是该表的看法。从表面上看，不难发现它是由十六进制数构成的。

例如，字符 '5' 位于第 3 列、第 5 排，那么它的字符编码就是十六进制的 35。

同理，字符 'A' 的字符编码是十六进制的 41。

那么，将字符 '0'、'1'、...、'9' 的值分别用十六进制数和十进制数表示就是：

	十六进制数	十进制数
'0'	30	48
'1'	31	49
'2'	32	50
'3'	33	51
:	:	:
'9'	39	57

表 8-1 JIS 编码表（按 C 语言风格显示）

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	'	p				一	タ	ミ		
1			!	1	A	Q	a	q				。	ア	チ	ム	
2			"	2	B	R	b	r				「	イ	ツ	メ	
3			#	3	C	S	c	s				」	ウ	テ	モ	
4			\$	4	D	T	d	t				、	エ	ト	ヤ	
5			%	5	E	U	e	u				.	オ	ナ	ユ	
6			&	6	F	V	f	v				ヲ	カ	ニ	ヨ	
7	\a		'	7	G	W	g	w				ア	キ	ヌ	ラ	
8	\b		(8	H	X	h	x				イ	ク	ネ	リ	
9	\t)	9	I	Y	i	y				ウ	ケ	ノ	ル	
A	\n		*	:	J	Z	j	z				エ	コ	ハ	レ	
B	\v		+	;	K	[k	{				オ	サ	ヒ	ロ	
C	\f		,	<	L	¥	l					ヤ	シ	フ	ワ	
D	\r		-	=	M]	m	}				ユ	ス	ヘ	ン	
E			.	>	N	^	n	~				ヨ	セ	ホ	ッ	
F			/	?	O	_	o					ッ	ソ	マ	°	

即数字字符 '0' ~ '9' 的字符编码用十进制表示为 48 ~ 57。

请注意这些值绝不是 0 ~ 9。字符 '0' 和数值 0 看起来相似，实则完全不同。

既然已经知道了数字字符 '0' ~ '9' 的值，那么

switch 语句就可以写成下图形式。

程序经过这样改写之后，其中的规律便显现了出来。

数字字符 *ch* 的值减去 48，即 *ch* - 48 得到的正好就是下标 0 ~ 9。

根据上述规律，我们可以将这个 **switch** 语句进一步简化为下图所示的 **if** 语句。

原来需要 10 多行的程序，现在仅用 2 行就能实现了！

但是，这个程序有个缺点，那就是缺乏可移植性。

目前为止我们谈论的字符相关内容，都是基于 JIS 码展开的。而在其他字符编码中字符 '0' 的值不一定是 48。

若在那样的环境中运行程序，就会错误地计算值为 48 ~ 57 的“其他字符”的出现次数，输出的结果也就不正确了。

不过幸运的是，C 语言程序的运行环境都遵循下面这一规则。

数字 '0'、'1'、……、'9' 的值是递增的

虽然 '0' 的值根据字符编码各有不同，但是无论在何种环境下，'5' 只会比 '0' 大 5。即 '5' - '0' 的值一定是 5 (专题 8-2)。

任意数字字符减去 '0'，都能得到所需的下标值。因此刚才的 **if** 语句可以改写成下图形式。

代码清单 8-9 所示的程序中使用了这段代码，这样看起来更为简洁。

```
switch (ch) {
    case 48: cnt[0]++; break;
    case 49: cnt[1]++; break;
    case 50: cnt[2]++; break;
    case 51: cnt[3]++; break;
    /*--- 中略 ---*/
    case 57: cnt[9]++; break;
}
```

```
if (ch >= 48 && ch <= 57)
    cnt[ch - 48]++;
```

```
if (ch >= '0' && ch <= '9')
    cnt[ch - '0']++;
```

专题 8-2 字符编码

如正文所述，C 语言中规定：

✓ 数字字符 '0'、'1'、……、'9' 的值是递增的

但是并不保证下面两条成立。

× 大写英文字母 'A'、'B'、……、'Z' 的值是递增的

× 小写英文字母 'a'、'b'、……、'z' 的值是递增的

例如，大型机中普遍使用的 EBCDIC 码就不遵循这个规则。

(当然，在 ASCII 码和 JIS 码中这个规则成立。)

代码清单 8-9

```

/*
  计算标准输入流中出现的数字字符数（第2版）
*/

#include <stdio.h>

int main(void)
{
    int i, ch;
    int cnt[10] = {0}; /* 数字字符的出现次数 */

    while (1) { /* 死循环 */
        ch = getchar();
        if (ch == EOF) break;

        if (ch >= '0' && ch <= '9')
            cnt[ch - '0']++;
    }

    puts("数字字符的出现次数");
    for (i = 0; i < 10; i++)
        printf("%d: %d\n", i, cnt[i]);

    return (0);
}

```

运行结果

```

3.1415926535897932846
Ctrl+D
数字字符的出现次数
'0': 0
'1': 2
'2': 2
'3': 3
'4': 2
'5': 3
'6': 2
'7': 1
'8': 2
'9': 3

```

程序变得相当简洁了。

我们通过代码清单 8-10 的程序，看看 EOF 和各个数字字符的值吧。虽然数字字符的值在不同运行环境下各有不同，但是程序本身并不依赖于运行环境。这点大家应该明白了吧？

代码清单 8-10

```

/*
  显示 EOF 和数字字符的值
*/

#include <stdio.h>

int main(void)
{
    int i;

    printf("EOF = %d\n", EOF);

    for (i = 0; i < 10; i++)
        printf("%d' = %d\n", i, '0' + i);

    return (0);
}

```

运行结果

```

EOF = -1
'0' = 48
'1' = 49
'2' = 50
'3' = 51
'4' = 52
'5' = 53
'6' = 54
'7' = 55
'8' = 56
'9' = 57

```

► 程序的运行结果依赖于编译器和运行环境。

转义字符

请看前面表 8-1 中的 JIS 码。位于 0x07 至 0x0D 的字符是：`\a`、`\b`、`\t`、`\n`、`\v`、`\f`、`\r`。其中，表示换行的 '`\n`' 和表示响铃的 '`\a`' 这两个转义字符，我们在第 1 章中已经学过了。表 8-2 中罗列了这些转义字符。

表 8-2 转义字符

简单转义字符 (simple escape sequence)		
<code>\a</code>	响铃 (alert)	发出警报声或显示警告
<code>\b</code>	退格符 (backspace)	光标左移一格
<code>\t</code>	水平制表符 (horizontal tab)	横向跳到下一制表位置
<code>\n</code>	换行符 (new line)	换行并移到下一行行首
<code>\v</code>	垂直制表符 (vertical tab)	纵向跳到下一制表位置
<code>\f</code>	换页符 (form feed)	换页并移到下一页页首
<code>\r</code>	回车符 (carriage return)	回到行首
<code>\'</code>	字符 '	单引号
<code>\"</code>	字符 "	双引号
<code>\?</code>	字符 ?	问号
<code>\\</code>	字符 \	反斜杠
八进制转义字符 (octal escape sequence)		
<code>\ccc</code>	ccc 为 1 ~ 3 位的八进制数。与 ccc 的值相对应的字符	
十六进制转义字符 (hexadecimal escape sequence)		
<code>\xhh</code>	hh 为任意位数的十六进制数。与 hh 的值相对应的字符	

字符 ' 的作用是将字符常量括起来，所以在表示该字符本身的时候不能写作 "，而应该使用转义字符 `\'`，写作 `'\'`。

八进制转义字符 (octal escape sequence) 和十六进制转义字符 (hexadecimal escape sequence) 可以用编码来表示任意字符。例如在 JIS 码中，数字字符 '1' 可以用 '`\61`' 或 '`\x31`' 来表示。不过这种表示方法会降低程序可移植性，所以尽量不要使用。

- ▶ 在 JIS 码中，字符可以用 8 位来表示，但也有些运行环境是用 9 位来表示的。所以在写程序时应注意不要假设字符总是 8 位的。
- 另外，考虑到 C 语言在有些环境下不能用 `char` 型来表示日语文字等字符集，从而制定出宽字符的概念。

● 练习 8-7

改写代码清单 8-9 的程序，将数字字符的出现次数用并排的 * 表示。

复制

数字字符计数程序会逐个读取输入的字符。将输入的字符按原样输出，实现了输入→输出的复制。

请看代码清单 8-11。

代码清单 8-11

```
/*
 从标准输入流复制到标准输出流
*/
#include <stdio.h>

int main(void)
{
    int ch;

    while ((ch = getchar()) != EOF)
        putchar(ch);

    return (0);
}
```

注意观察 **while** 语句的循环条件表达式。

首先，如图 8-6 (a) 所示，将读入的字符赋给 *ch*，*ch* 就与赋值表达式 *ch=getchar()* 的值相等。

► 赋值表达式的功能是计算表达式的值再赋予左边的变量（5-1 节）。

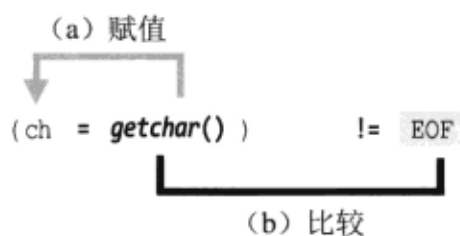


图 8-6 while 语句循环条件表达式的说明

接着，如图 8-6 (b) 所示，将 *ch=getchar()* 的值和 EOF 作比较。若运算符 **!=** 两边的值不相等，则表达式的值为 1，反之为 0（3-1 节）。

由此可知，这个 **while** 语句的循环结构是和之前的计数程序完全相同的。C 语言代码竟可以写得如此简洁！

在 **while** 语句循环内部，调用了 **putchar** 函数。它的功能是输出参数中的字符。下面有该函数的详细说明。

综上所述，该程序会不断地将读入的字符逐一输出到显示器上，直至输入结束或者发生错误为止。

putchar	
头文件	#include <stdio.h>
原型	int putchar(int c)
说明	向标准输出流中输出字符 c。
返回值	返回输出的字符。发生错误时，返回 EOF。

下面是程序运行结果。

```

运行结果
Hello!
Hello!
This is a pen.
This is a pen.
Ctrl + Z

```

► 运行结果依赖于提供程序运行环境的操作系统等。

上图是在 MS-DOS (MS-Windows) 上的运行结果。其中，Ctrl + Z 意为输入结束，操作方法是按住 Ctrl 键不放，同时按下 Z 键。而在 UNIX 中，则使用 Ctrl + D。

本例所示程序并不是每读入一个字符后就马上输出，而是在按下 键后一并输出。C 语言的输入输出一般会将读入的字符以及待输出的字符暂时保存在缓存中，当达到下列条件时才进行实际的输入输出操作。

“缓存已满” 全缓冲
或

“读入换行符” 行缓冲

当然，也有下面这样的环境。

“立即输出” 无缓冲

而在下述命令中，给定输入和输出文件名（假设运行文件的名称为 CPY.EXE）并运行：

```

A> CPY < 输入文件名 > 输出文件名

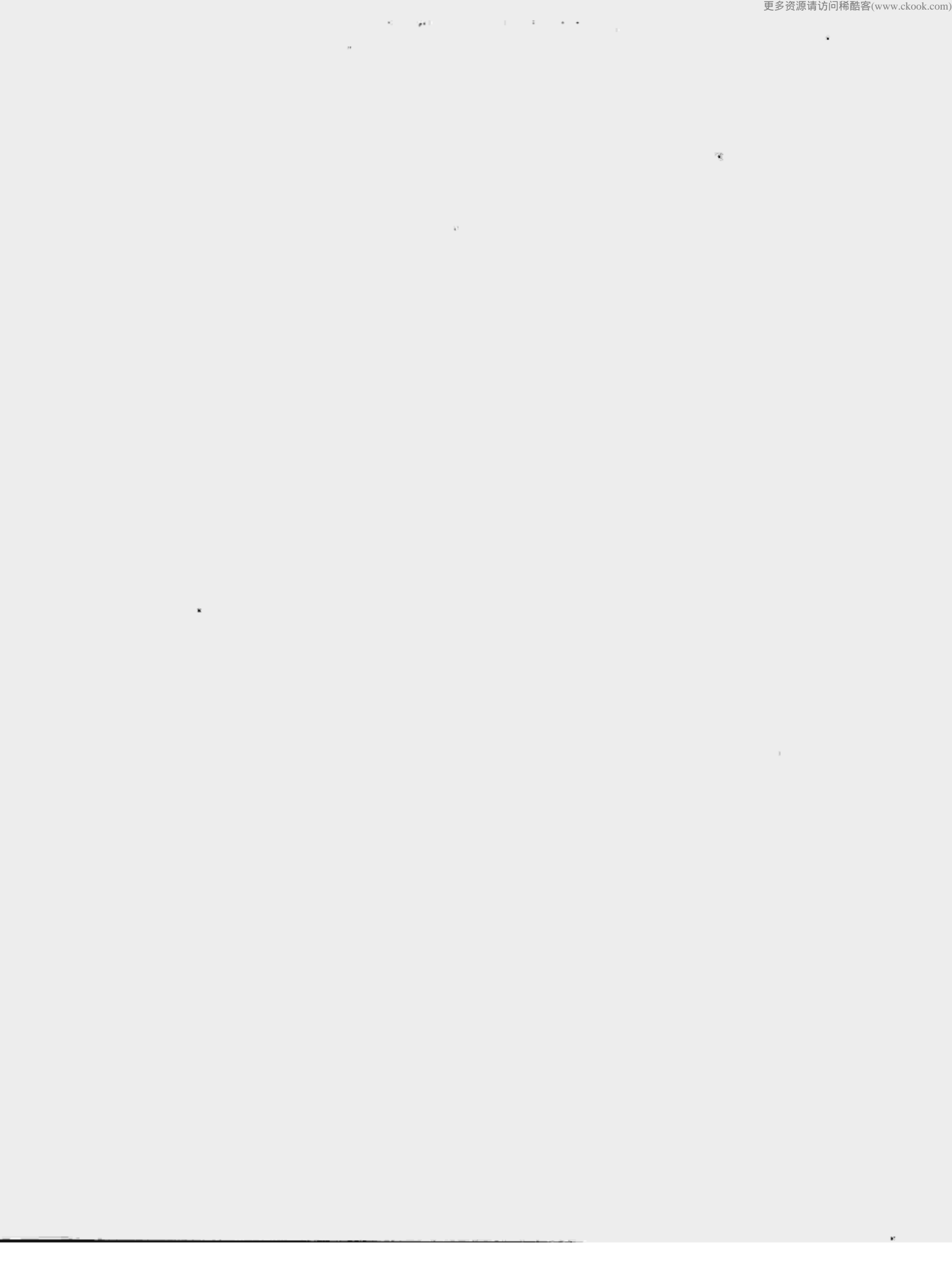
```

“输入文件”的数据就会复制到“输出文件”中去。但这不是由 C 语言实现的，而是通过 UNIX 和 MS-DOS 等操作系统的重定向功能来实现。

至此，我们已经了解到 printf、puts、putchar 函数会将数据显示到“显示器”，scanf 函数则会从“键盘”读入数据。然而，输入输出对象并不仅限于这些设备。我们将前者的输出对象称为标准输出流，后者的输入源称为标准输入流。

● 练习 8-8

编写程序，计算标准输入流中的出现的行数。



第 9 章

字符串的基本知识

我们在上一章的后半部分学习了字符的有关内容。不过，环顾四周你就会发现仅用一个字符就能表示的事物少之又少，例如名字、地名等，在绝大多数情况下都是由多个字符组成的。

本章我们就将学习字符序列——字符串的基本知识。



9-1 什么是字符串

字符串字面量

像 **"ABC"** 那样带双引号的一系列字符称为**字符串字面量**。

在字符串字面量的末尾会被加上一个叫做 **null** 字符的值为 0 的字符。用八进制转义字符表示 null 字符就是 **'\0'**。

► null 字符的每一位都是 0。若不用字符常量，而用整数常量表示就是 0。

因此，字符串字面量 **"ABC"** 实际上占用了 4 个字符的内存空间，如图 9-1 (a)。而双引号中没有任何字符的字符串字面量 **""** 表示的就是 null 字符，如图 9-1 (b)。

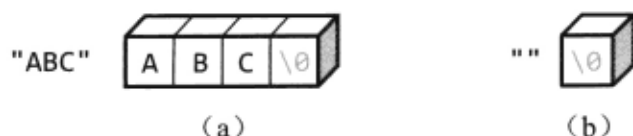


图 9-1 字符串字面量的内部实现

如果要在字符串字面量中表示双引号 **"**，就需要使用转义字符 ****。例如要表示 **xy"z** 的字符串字面量就必须写作 **"xy\"z"**。

字符串字面量的长度

我们可以编写程序来分别显示三个字符串字面量的长度，以此来确认字符串字面量的末尾被加上了 null 字符。请见代码清单 9-1 所示程序。

这三个字符串字面量在内存中的存储形式如图 9-2 所示。

通过运行结果可知，运行环境会忽略字符串字面量 **"abc\0def"** 中间的 null 字符 **"\0"**，而另行在末尾加上一个 **null** 字符。

► 字符串字面量 **"AB\tC"** 中的 **'\t'** 表面上是两个字符，实际上是表示水平制表的转义字符，因此算作一个字符。

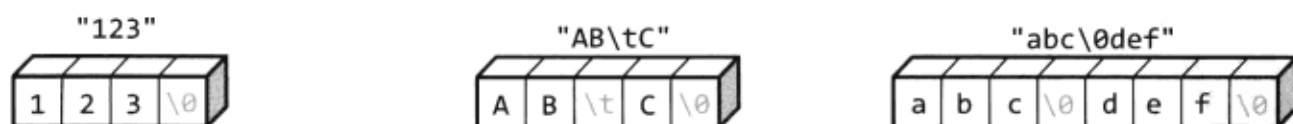


图 9-2 字符串字面量的长度

代码清单 9-1

```

/*
  显示字符串字面量长度
*/

#include <stdio.h>

int main(void)
{
    printf("sizeof(\"123\")      = %u\n",      (unsigned)sizeof("123"));
    printf("sizeof(\"AB\\tC\")    = %u\n",      (unsigned)sizeof("AB\\tC"));
    printf("sizeof(\"abc\\0def\") = %u\n",      (unsigned)sizeof("abc\\0def"));

    return (0);
}

```

运行结果

```

sizeof("123")      = 4
sizeof("AB\\tC")    = 5
sizeof("abc\\0def") = 8

```

► 我们在第 7 章学习了 **sizeof** 运算符。

下面我们来总结一下字符串字面量的性质。

具有静态生命周期

右图所示函数的功能是显示两次 "ABCD"。调用该函数时必须将字符串字面量 "ABCD" 传入 **puts** 函数。因此字符串字面量 "ABCD" 就必须“活在”程序开始到结束的整个生命周期内。

正因如此，字符串字面量自然被赋予了静态生命周期。

```

void func(void)
{
    puts("ABCD");
    puts("ABCD");
}

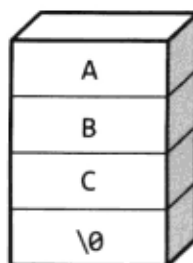
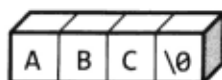
```

对于同一字符串字面量的处理方式依赖于编译器

func 函数中有两个拼写完全相同的字符串字面量 "ABCD"。如果将它们视为相同，并共用同一个字符串字面量，就能减少所需内存空间（由 10 个字符减少到 5 个字符）。

有些编译器会共用这种拼写相同的字符串字面量，具体请查阅你使用的编译器的说明文档。

► 本书中的字符串字面量和字符串的图示有时是纵向的，有时是横向的。这只是为了节省图示空间，并无特别用意。



字符串

字符串字面量类似于整数的 50、浮点数的 3.14 等常量。数值型数据可以通过变量（对象）的数据类型转换进行混合运算。而表示字符序列的字符串（string）也可以以对象形式保存并灵活处理。

C 语言中的字符串是以字符序列，即字符数组实现的。例如要表示字符串 "ABC"，数组元素必须按以下顺序依次保存（图 9-3）。

'A'、'B'、'C'、'\0'

末尾的 null 字符 '\0' 是字符串结束的“标志”。

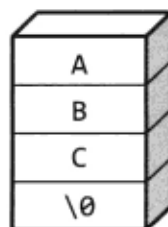


图 9-3 字符串

■ 注意 ■

字符串是以字符数组的形式实现的。其末尾是首次出现的 null 字符。

► 字符串字面量的中间也可以有 null 字符，不过应注意区分。字符串字面量 "ABC" 是字符串，而字符串字面量 "AB\0CD" 不是字符串。

以字符数组的形式保存并显示字符串 "ABC" 的程序如代码清单 9-2 所示。

代码清单 9-2

```
/*
保存字符串的数组
*/

#include <stdio.h>

int main(void)
{
    char str[4];          /* 保存字符串的数组 */

    str[0] = 'A';         /* 赋值 */
    str[1] = 'B';         /* 赋值 */
    str[2] = 'C';         /* 赋值 */
    str[3] = '\0';        /* 赋值 */

    printf("字符串 str 为%s.\n", str); /* 显示 */

    return (0);
}
```

运行结果

字符串 str 为 ABC。

上述表示字符串的转换说明为 "%s"。

► 转换说明中的 s 是字符串 string 的缩写。

字符数组的初始化赋值

为保存字符串而将每个字符逐一赋予数组的各个元素可不是一件轻松的事，所以我们可以这样进行声明：**char str[4] = {'A', 'B', 'C', '\0'};**

这样不仅简洁，而且也能确保数组元素的初始化，且在形式上与 **int** 型、**double** 型等数组的初始化完全一致。该声明还能进一步简化为：**char str[4] = "ABC";**

通常这种声明方式更为简洁也更常用。

■ 注意 ■

以下两种形式都可以实现字符数组的初始化赋值。

(a) **char ss[] = {'A', 'B', 'C', '\0'};**

(b) **char ss[] = "ABC";**

► 以上声明中，元素数都能省略（此时，数组 *ss* 的元素数视为 4）。此外，(b) 的初始值也可以用 {} 括起来，如 {"ABC"}。

我们来改写一下上一页的程序，不将字符逐一赋予数组的各个元素，而采用初始化赋值的方法。请见代码清单 9-3。

代码清单 9-3

```
/*
 保存字符串的数组（初始化）
*/
#include <stdio.h>

int main(void)
{
    char str[] = "ABC";    /* 初始化 */
    printf("字符串 str 为 %s。 \n", str); /* 显示 */
    return (0);
}
```

运行结果

字符串 str 为 ABC。

但是除了初始化赋值的时候，我们不能将数组的初始值或字符串直接赋予数组变量。

str = {'A', 'B', 'C', '\0'}; /* 错误：不能赋值 */

str = "ABC"; /* 错误：不能赋值 */

● 练习 9-1

将代码清单 9-3 中数组 *str* 的声明改为：

char str[] = "ABC\0DEF"

查看程序的运行结果。

空字符串

内容为空，即开头就是 `null` 字符的字符串称为**空字符串**（`null string`）（图 9-4）。可以通过下述语句声明一个保存空字符串的数组（数组的元素数为 1 个）。

```
char ns[] = " ";
```

► 当然，也可以进行如下声明，效果相同。

```
char ns[] = {'\0'};
```

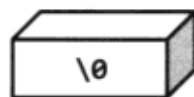


图 9-4 空字符串

字符串的读取

代码清单 9-4 所示程序的功能是读取一个表示名字的字符串，并显示问候语。

代码清单 9-4

```
/*
 * 询问名字并显示问候语（读取字符串）
 */

#include <stdio.h>

int main(void)
{
    char name[40];

    printf("请输入您的名字: ");
    scanf("%s", name);

    printf("您好, %s 先生 / 女士 !!\n", name);

    return (0);
}
```

运行结果

请输入您的名字: Shibata
您好, Shibata 先生 / 女士 !!

我们事前无法确定输入的名字会有多少个字符，因此数组的元素数必须能容纳足够多的字符。本例假设为 40 个元素（由于末尾 `null` 字符会占去 1 个元素，所以实际可容纳 39 个字符）。

为了从标准输入读取字符串，需要把 `scanf` 函数的转换说明设为 `"%s"`，还必须传入数组 `name`。请注意这里的 `name` 是不带 `&` 运算符的。

● 练习 9-2

如何让下述初始化赋值得到的字符串 `str` 变成空字符串？请编写程序实现。

```
char str[] = "ABC";
```

格式化显示字符串

第 2 章中简单介绍了表示整数和浮点数的转换说明。字符串也是通过这种方式进行说明的。

请见代码清单 9-5 中的几个例子。

代码清单 9-5

<pre> /* 格式化字符串 "12345" 并显示 */ #include <stdio.h> int main(void) { char str[] = "12345"; printf("%s\n", str); /* 原样输出 */ printf("%3s\n", str); /* 至少显示 3 位 */ printf("%.3s\n", str); /* 最多显示 3 位 */ printf("%8s\n", str); /* 右对齐 */ printf("%-8s\n", str); /* 左对齐 */ return (0); } </pre>	<div>运行结果</div> <pre> 12345 12345 123 12345 12345 </pre>
--	--

格式控制字符串为如下形式：

%9.9s

(a) 输出最小宽度

表示至少要输出指定的位数。

如果省略本项或实际输出的字符串位数超过指定值，则按实际位数输出。

如果设置了 - 标志，则表示左对齐，否则表示右对齐（空白部分填补空格）。

(b) 精度

指定显示位数的上限（即不可能显示超过指定位数的字符，超过则截去）。

(c) 转换说明符

s 表示输出字符串。即输出数组的字符，直到 `null` 字符的前一个字符为止。

如果没有指定精度或精度大于数组长度，则数组中必须含有 `null` 字符。

► 这里介绍的转换说明只是冰山一角。详细内容请见附录 2。

9-2 字符串数组

字符串数组

类型相同的数据集合适合用数组来实现。当然，用数组来保存多个字符串也是可以的。

代码清单 9-6 所示的程序中，二维数组 *cs* 的 3 个元素 *cs*[0]、*cs*[1]、*cs*[2] 分别初始化为字符串 "Turbo"、"NA"、"DOHC"。

代码清单 9-6

```
/*
 * 字符串数组
 */

#include <stdio.h>

int main(void)
{
    int    i;
    char  cs[][6] = {"Turbo", "NA", "DOHC"};

    for (i = 0; i < 3; i++)
        printf("cs[%d] = \"%s\"\n", i, cs[i]);

    return (0);
}
```

运行结果

```
cs[0] = "Turbo"
cs[1] = "NA"
cs[2] = "DOHC"
```

► 在二维数组的声明中，只要给定初始值，第一维的元素数就可以省略。

通过该声明，数组 *cs*[0] 表示 "Turbo"、*cs*[1] 表示 "NA"、*cs*[2] 表示 "DOHC"。它们在内存中的存储形式如图 9-5 所示。

二维数组的各个元素都由两个下标来表示。'T' 为 *cs*[0][0]、'c' 为 *cs*[2][3]。

<i>cs</i> [0]	T	u	r	b	o	\0
<i>cs</i> [1]	N	A	\0	\0	\0	\0
<i>cs</i> [2]	D	O	H	C	\0	\0

► 数组声明中初始值不足时，用 0 来初始化该元素。因此各个字符串后面的空白部分都初始化为 null 字符。

图 9-5 由二维数组实现的字符串数组

读取字符串数组中的字符串

代码清单 9-7 所示程序的功能是，将从标准输入读到的字符串的各个字符逐个往字符串数组中的各个元素赋值。

代码清单 9-7

```
/*
  读取并显示字符串数组
*/

#include <stdio.h>

int main(void)
{
    int    i;
    char   cs[3][128];

    for (i = 0; i < 3; i++) {
        printf("cs[%d]: ", i);
        scanf("%s", cs[i]);
    }

    for (i = 0; i < 3; i++)
        printf("cs[%d] = \"%s\\n\"", i, cs[i]);

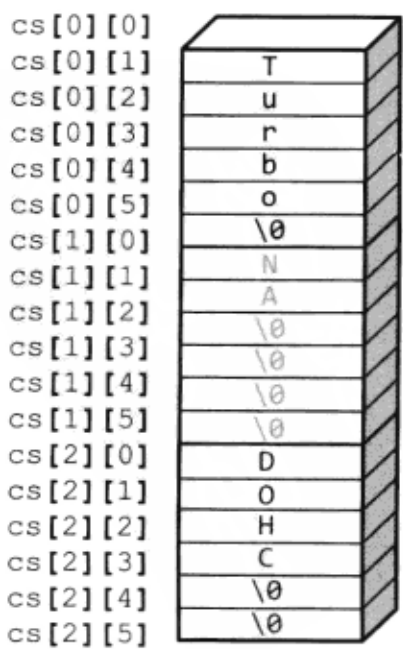
    return (0);
}
```

运行结果

cs[0]: S500L
cs[1]: A8
cs[2]: SLK
cs[0] = "S500L"
cs[1] = "A8"
cs[2] = "SLK"

因为 cs[0]、cs[1]、cs[2] 都是字符串（字符数组），所以将它们传入 scanf 函数时不可以带 & 运算符。

► 二维数组看上去是二维表，但在物理内存中是线性排列的。因此，代码清单 9-6 中的数组 cs 在内存中的存储形式如右图所示。



9-3 字符串处理

字符串长度

我们来对下述语句声明的数组 *str* 进行思考。

```
char str[6] = "ABC";
```

如图 9-6 所示，元素数为 6 的数组中保存了长度为 3（算上字符串末尾的 null 字符，则长度为 4）的字符串。因此，实际上数组的后几个元素都是空的。

由此可知，字符串不一定正好撑满字符数组。

因为字符串中含有表示其末尾的 null 字符，所以第一个字符到 '\0'（的前一个字符）为止的字符数，就是该字符串的长度。

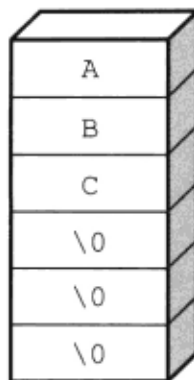


图 9-6 字符数组及其中的字符串

基于以上思路，我们可以写出计算字符串长度的程序，请见代码清单 9-8。

代码清单 9-8

```
/*
 * 判断字符串的长度
 */
#include <stdio.h>

/* 返回字符串 str 的长度 */
unsigned str_length(const char str[])
{
    unsigned len = 0;

    while (str[len])
        len++;

    return (len);
}

int main(void)
{
    char st[100];

    printf("请输入字符串: ");
    scanf("%s", st);

    printf("字符串 %s 的长度为 %u。 \n", st, str_length(st));

    return (0);
}
```

运行结果

请输入字符串: SEC
字符串 SEC 的长度为 3。

先来看一下 **main** 函数调用 `str_length` 函数时传入的实参 `st`。实参只要给出数组名称就行了，这和传入 **int** 型数组时的情况是一样的。

因此在 **main** 函数中定义的数组 `st`，可以在 `str_length` 函数中作为数组 `str` 进行处理。

接着，在 `str_length` 函数中使用变量 `len`，遍历数组计算字符串的长度。

注意观察程序中的 **while** 语句。**while** 语句在循环条件表达式为非 0 的情况下，会执行循环体语句。该循环语句的说明如下。

```
while (str[len] != 0)    /* while (str[len] != '\0') */
    len++;              /*      len++;              */
```

变量 `len` 的初始值为 0，每次执行循环体语句就自增 1，直至出现 null 字符为止（图 9-7）。

那么，当 **while** 语句结束时，变量 `len` 的值就是字符串的长度。

当然，也可以将 `str_length` 函数想作“返回数组 `str` 中首个值为 null 的元素的下标值的函数”。

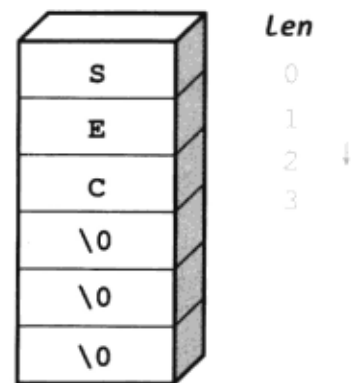


图 9-7 求出字符串的长度

● 练习 9-3

编写如下函数，若字符串 `str` 中含有字符 `c`（若含有多个，以先出现的为准），则返回该元素的下标值，否则返回 -1。

```
int str_char(const char str[], int c) { /* ... */ }
```

● 练习 9-4

编写如下函数，返回字符串 `str` 中字符 `c` 的个数（没有则返回 0）

```
int str_chnum(const char str[], int c) { /* ... */ }
```

遍历字符串

这次我们不使用 **printf** 函数和 **puts** 函数，而只使用 **putchar** 函数来显示字符串。可以通过对每个字符进行遍历来实现。请见代码清单 9-9 所示的程序。

代码清单 9-9

```
/*
 * 依次显示字符串中的字符
 */

#include <stdio.h>

/* 显示字符串（不换行）*/
void put_string(const char str[])
{
    unsigned i = 0;

    while (str[i])
        putchar(str[i++]);
}


int main(void)
{
    char str[100];

    printf("请输入字符串:");
    scanf("%s", str);

    put_string(str);
    putchar('\n');

    return (0);
}
```

运行结果

请输入字符串: SEC 
SEC

对字符串中的字符进行遍历的步骤和上一页的 `str_length` 函数一样。对每个字符进行遍历，直到出现 `null` 字符为止。

● 练习 9-5

改写代码清单 9-9 中的 `put_string` 函数，不使用 **putchar** 函数，而改用 **printf** 函数实现同样功能。

● 练习 9-6

编写如下函数，使字符串 `str` 显示 `no` 次。

```
void put_stringn(const char str[], int no) { /* ... */ }
```

数字字符的出现次数

代码清单 9-10 所示程序的功能是遍历字符串中的各个字符，并计算其中数字字符 '0' ~ '9' 的个数。

代码清单 9-10

```
/*
 * 计算字符串中的数字字符数
 */

#include <stdio.h>

/*--- 将字符串 str 中的数字字符保存至数组 cnt---*/
void str_dcount(const char str[], int cnt[])
{
    unsigned i = 0;
    while (str[i]) {
        if (str[i] >= '0' && str[i] <= '9')
            cnt[str[i] - '0']++;
        i++;
    }
}

int main(void)
{
    int i;
    int dcnt[10] = {0};
    char str[100];

    printf("请输入字符串: ");
    scanf("%s", str);

    str_dcount(str, dcnt);

    puts("数字字符的出现次数");
    for (i = 0; i < 10; i++)
        printf("%d': %d\n", i, dcnt[i]);

    return (0);
}
```

运行结果

请输入字符串:

3.1415926535897932846

数字字符的出现次数

'0': 0

'1': 2

'2': 2

'3': 3

'4': 2

'5': 3

'6': 2

'7': 1

'8': 2

'9': 3

● 练习 9-7

根据代码清单 9-9 的函数，编写 `put_rstring` 函数，实现字符串的逆向输出（如将 "SEC" 显示为 "CES"）。

● 练习 9-8

编写如下函数，实现字符串 `str` 的逆向保存（如将 "SEC" 保存为 "CES"）。

```
void rev_stringn(char str[]) { /* ... */ }
```

字符串数组的参数传递

我们来编写一个程序，在函数之间传递用二维数组实现的“字符串数组”。参数传递方法和普通的 `int` 型等二维数组相同。

我们将代码清单 9-6 中显示字符串数组的程序改写了一下，这次使用了函数调用。请见代码清单 9-11。

代码清单 9-11

```
/*
 * 显示字符串数组（函数版）
 */

#include <stdio.h>

/*--- 显示字符串数组 ---*/
void put_strary(const char st[][6], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("st[%d] = \"%s\\\"\\n", i, st[i]);
}

int main(void)
{
    char cs[][6] = {"Turbo", "NA", "DOHC"};

    put_strary(cs, 3);

    return (0);
}
```

运行结果

```
st[0] = "Turbo"
st[1] = "NA"
st[2] = "DOHC"
```

- 在接受二维数组的形参的声明中，只有第一维的数组元素数可以省略。因此下面这样的声明是不正确的。

```
void put_strary(const char st[][], int n)
```

而 `put_strary` 函数的意思是只能接收元素数为 6 的字符串（字符数组）数组。

二维数组的各个元素，可以使用两个下标运算符来表示。

这样，通过对字符串中的每个元素（字符）进行遍历，也可以将字符串显示出来。请见代码清单 9-12。

代码清单 9-12

```

/*
 显示字符串数组
*/

#include <stdio.h>

/*--- 显示字符串数组（逐个显示字符）---*/
void put_strary2(const char st[][6], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        int j = 0;
        printf("st[%d] = \", i);

        while (st[i][j])
            putchar(st[i][j++]);

        puts("\n");
    }
}

int main(void)
{
    char cs[][6] = {"Turbo", "NA", "DOHC"};

    put_strary2(cs, 3);

    return (0);
}

```

运行结果

```

st[0] = "Turbo"
st[1] = "NA"
st[2] = "DOHC"

```

专题 9-1 字符串的初始化

在前面我们已经介绍过可以使用下列方式对保存字符串 "ABC" 的字符数组进行初始化。

```
char str[4] = "ABC";
```

那么如果将这条声明语句改写一下会如何呢？

```
char str[4] = "ABCD";
```

这样一来，算上 null 字符需要 5 个字符的空间，但数组只能接收 4 个字符。

事实上，若像下面这样进行声明，末尾就不会加上 null 字符。即：

```
char str[4] = {'A', 'B', 'C', 'D'};
```

这样声明的变量不会被当作字符串，我们把它当作 4 个字符的集合，也就是“普通的”数组来使用就行了。

大小写字符转换

我们来编写两个函数，一个是将字符串中的英文字符全部转为大写字母，另一个则全部转为小写字母。请见代码清单 9-13。

代码清单 9-13

```
/*
 对字符串中的英文字符进行大小写转换
*/

#include <ctype.h>
#include <stdio.h>

/*--- 将字符串中的英文字符转为大写字母 ---*/
void str_toupper(char str[])
{
    unsigned i = 0;
    while (str[i]) {
        str[i] = toupper(str[i]);
        i++;
    }
}

/*--- 将字符串中的英文字符转为小写字母 ---*/
void str_tolower(char str[])
{
    unsigned i = 0;
    while (str[i]) {
        str[i] = tolower(str[i]);
        i++;
    }
}

int main(void)
{
    char str[100];

    printf("请输入字符串:");
    scanf("%s", str);

    str_toupper(str);
    printf("大写字母: %s\n", str);

    str_tolower(str);
    printf("小写字母: %s\n", str);

    return (0);
}
```

运行结果

请输入字符串: BohYoh 
 大写字母: BOHYOH
 小写字母: bohyoh

- 如果在被调函数中修改传入的数组元素值，那么这个修改也会反映到主调函数的数组中。

toupper**头文件** `#include <ctype.h>`**原 型** `int toupper(int c)`**说 明** 将小写英文字母转换为相应的大写英文字母。**返回值** 若 *c* 是小写英文字母，则返回转换后的大写字母，否则直接返回 *c*。**tolower****头文件** `#include <ctype.h>`**原 型** `int tolower(int c)`**说 明** 将大写英文字母转换为相应的小写英文字母。**返回值** 若 *c* 是大写英文字母，则返回转换后的小写字母，否则直接返回 *c*。**● 练习 9-9**

编写如下函数，将字符串 *str* 转换为空字符串。

```
void null_string(char str[]) { /* ... */ }
```

● 练习 9-10

编写如下函数，将字符串 *str* 中的数字字符全部删除。（例如传入 "AB1C9" 则返回 "ABC"）

```
void del_digit(char str[]) { /* ... */ }
```



第 10 章

指 针

我们在学习过程中，对学习对象的思考方式无时无刻都在变化，这一点不仅限于编程。

通过本章的学习，我们将把保存数据的“魔术盒”——变量（对象）作为占据一部分内存空间的对象来重新认识。我们终于要敲开“指针”的大门了，这是 C 语言学习过程中要攻克的难关之一。



10-1 指 针

函数的参数

代码清单 10-1 中的程序是用来计算两个整数的和与差的。

`sum_diff` 函数会求出参数 `n1` 和 `n2` 的和与差。但在调用 `sum_diff` 函数之后, `main` 函数中初始化为 0 的 `wa` 和 `sa` 的值却依然是 0, 不能获得预期结果。

代码清单 10-1

```
/*
 * 计算两个整数的和与差 (误例)
 */

#include <stdio.h>

/*--- 将 n1 和 n2 的和、差分别保存至 sum、diff (误例) ---*/
void sum_diff(int n1, int n2, int sum, int diff)
{
    sum = n1 + n2;
    diff = (n1 > n2) ? n1 - n2 : n2 - n1;
}

int main(void)
{
    int na, nb;
    int wa = 0, sa = 0;

    puts("请输入两个整数。");
    printf("整数 A:"); scanf("%d", &na);
    printf("整数 B:"); scanf("%d", &nb);

    sum_diff(na, nb, wa, sa);

    printf("两数之和是 %d。 \n 两数之差是 %d。 \n", wa, sa);

    return (0);
}
```

运行结果

请输入两个整数。
整数 A: 57
整数 B: 21
两数之和是 0。
两数之差是 0。

`main` 函数调用 `sum_diff` 函数时, 实参 `na`、`nb`、`wa`、`sa` 的值会分别传给形参 `n1`、`n2`、`sum`、`diff`。这个复制过程是单向的, 这种参数传递方式称为**值传递**。

这样即使改变 `sum_diff` 函数中的形参 `sum`、`diff` 的值, 原来的 `wa`、`sa` 也不会发生任何变化。

为了解决这个问题, 必须掌握 C 语言学习的难点之一——**指针 (pointer)**。本章将学习指针的基础知识。

变量和对象

变量是“保存数值的盒子”，它并不是像图 10-1(a) 那样无序存放的，而是如图 10-1(b) 所示那样有序地排列在内存空间里。

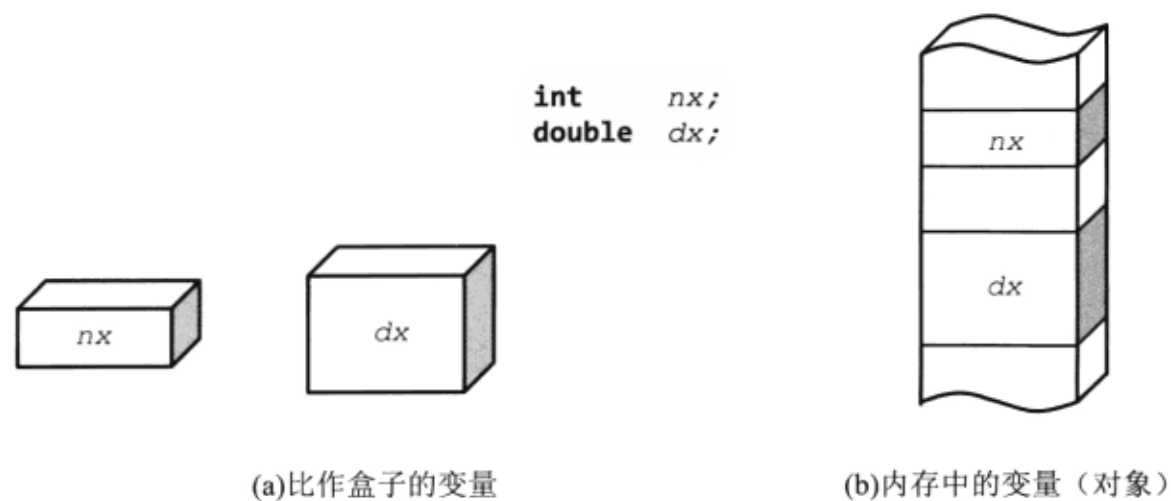


图 10-1 对象

“变量”具有多个侧面或者说多个属性。举例来说，其中一个属性就是数据类型长度。图中 `int` 型变量 `nx` 和 `double` 型变量 `dx` 就具有不同的长度。

► 当然，在有些编译器中 `sizeof(int)` 和 `sizeof(double)` 是相等的，但是构成它们的每一位的意义却不尽相同，这在第 7 章中已作说明。

数据类型决定了变量可以表示的数值范围。而该属性正是反映了数据类型的概念。除此之外，表示变量在内存中生命期范围的作用域（第 6 章）以及 `nx`、`dx` 等变量名也都是变量的重要属性。

在第 2 章中出现过的对象（object）也具备多个性质和属性。

地址

地址（address）表示对象在内存中的“位置”。英语单词 address 可以表示“演说”、“地址”等意思。这里我们不妨把它理解为“地址”。因为它正好和地址中的门牌号是一样的。

■ 注意 ■

对象的地址是指对象在内存中的存储位置编号。

取址运算符

每个对象都有地址，那么我们来看看地址究竟是怎么样的。请见代码清单 10-2 所示程序。

代码清单 10-2

```
/*
 * 显示对象的地址
 */

#include <stdio.h>

int main(void)
{
    int    nx;
    double dx;
    int    vc[3];

    printf("nx    的地址: %p\n", &nx);
    printf("dx    的地址: %p\n", &dx);
    printf("vc[0] 的地址: %p\n", &vc[0]);
    printf("vc[1] 的地址: %p\n", &vc[1]);
    printf("vc[2] 的地址: %p\n", &vc[2]);

    return (0);
}
```

运行结果	
nx	的地址: FFC D
dx	的地址: FFC 2
vc[0]	的地址: FFB 0
vc[1]	的地址: FFB 2
vc[2]	的地址: FFB 4

► 对象的地址通常是用十六进制数表示的。但是不同的编译器或不同的运行环境下，基数、位数等显示形式以及具体数值都会有所不同。

我们一直使用的单目运算符 &（unary & operator）通常被称为取址运算符（address operator）。将 & 运算符写在对象名之前，就可以得到该对象的地址（表 10-1）。如果对象的长度为 2，占用了 100 号和 101 号内存单元，那么该对象的地址就是它的首地址 100 号。

表 10-1 单目运算符 &（取址运算符）

单目运算符	&a	取得 a 的地址（生成指向 a 的指针）。
-------	----	-----------------------

► 双目运算符 & 为第 7 章中出现的按位与（AND）逻辑运算符。

注意

取址运算符 & 的功能是取得对象的地址。

表示对象地址的转换说明为 "%p"。

► 转换说明 "%p" 中的 p，是 pointer 的首字母。

观察运行结果可知，因为数组 vc 中各元素的长度就是 sizeof(int) 所得的值（本书假定该值为 2），所以该数组中相邻元素的首地址的间距即为 sizeof(int)。

指针

只显示对象的地址没有太大的用处，我们还是来看代码清单 10-3 中更具实际作用的程序吧。

代码清单 10-3

```
/*
 通过指针间接地操作身高
*/

#include <stdio.h>

int main(void)
{
    int    sato    = 178;    /* 佐藤的身高 */
    int    sanaka  = 175;    /* 佐中的身高 */
    int    hiraki  = 165;    /* 平木的身高 */
    int    masaki  = 179;    /* 真崎的身高 */

    int *isako, *hiroko;

    isako = &sato;           /* isako 指向 sato (喜欢佐藤) */
    hiroko = &masaki;        /* hiroko 指向 masaki (喜欢真崎) */

    printf("伊沙子喜欢的人的身高: %d\n", *isako);
    printf("洋子喜欢的人的身高: %d\n", *hiroko);

    isako = &sanaka;        /* isako 指向 sanaka (移情别恋) */

    *hiroko = 180;          /* 将 hiroko 指向的对象赋为 180 */
                           /* 修改洋子喜欢的人的身高 */

    putchar('\n');
    printf("佐藤的身高: %d\n", sato);
    printf("佐中的身高: %d\n", sanaka);
    printf("平木的身高: %d\n", hiraki);
    printf("真崎的身高: %d\n", masaki);
    printf("伊沙子喜欢的人的身高: %d\n", *isako);
    printf("洋子喜欢的人的身高: %d\n", *hiroko);

    return (0);
}
```

► 程序的运行结果见后文。

在变量 *isako* 和 *hiroko* 的声明中，变量名前带有 *。通过该声明定义了两个“指向 **int** 型变量的指针变量”，它们指向的是 **int** 型对象。

► 通过以下声明定义的 *hiroko* 不是指针变量，而是整型变量。

```
int *isako, hiroko;           /* isako 是指针变量, hiroko 是整型变量 */

要声明多个指针变量，应该分别在变量名前加上 *。

int *isako, *hiroko;         /* isako 和 hiroko 都是指针变量 */
```

我们首先明确一下“**int** 型变量”和“指向 **int** 型变量的指针变量”有什么区别。

int 型变量:

保存“整数”的盒子。

指向 int 型变量的指针变量:

保存“存放整数对象的地址”的盒子。

指针 *isako* 和 *hiroko* 并不保存一般的“整数”值，而是保存“存放整数的对象的地址”。

图 10-2 中，*sato* 的值为 178，而它的地址是 100 号。此处若如下赋值，

```
isako = &sato;
```

则 *isako* 的值就是“*sato* 的地址 100 号”。

这时，我们称 *isako* 指向 *sato*。

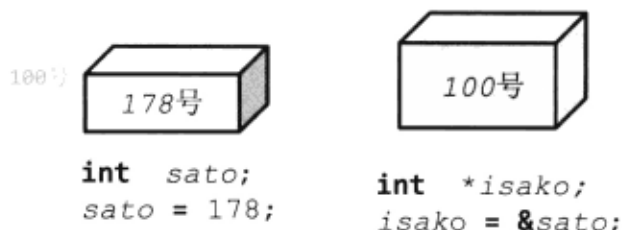


图 10-2 int 型变量和指向 int 型变量的指针变量

■ 注意 ■

当指针 *ptr* 的值为对象 *x* 的地址时，一般说 *ptr* 指向 *x*。

“指向”这一表述比较抽象，在这里可以理解成：

isako 喜欢 *sato* ①♥

接着进行 *hiroko* = &*masaki* 的赋值，那就可以得出：

hiroko 喜欢 *masaki* ♥

我们可以用图 10-3 来表示指针指向对象的情形。这里箭头指向的是喜欢的人。



图 10-3 指针

► *isako* 的数据类型是“指向 int 型变量的指针型”。

```
isako = &sato;
```

从以上赋值语句亦可发现，&*sato* 的类型也是“指向 int 型变量的指针型”。取址运算符与其说是取得地址，不如说是生成指针。表达式 &*sato* 是指向 *sato* 的指针，其值为 *sato* 的地址。

① 作者将变量进行了拟人化，文中以下变量名皆为日本人名。*isako*（伊沙子，女）、*hiroko*（洋子，女）、*sato*（佐藤，男）、*masaki*（真崎，男）、*sanaka*（佐中，男）、*hiraki*（平木，男）。

指针运算符

在进行显示的地方，就要用到叫做**指针运算符**^① (indirect operator) 的单目运算符 ***** (unary ***** operator) 了。

将指针运算符 ***** 写于指针之前，`printf("伊沙子喜欢的人的身高: %d\n", *isako);` 就可以显示该指针指向的对象内容。

(表 10-2)。

因此，`*isako` 就等于“`isako` 指向的对象 (伊沙子喜欢的男子的身高)”。

即 `*isako` 就是 `sato`。

`*isako` 是 `sato` 的别名 (alias)。

本书使用图 10-4 这种形式来表示指针。用虚线与对象连接的盒子中写有名字，这就是对象的别名。

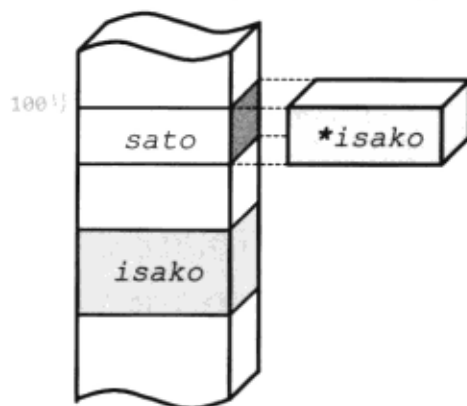


图 10-4 指针和别名

■ 注意 ■

当 `ptr` 指向 `x` 时，`*ptr` 就是 `x` 的别名。

■ 表 10-2 单目运算符 ***** (指针运算符)

单目运算符	*	*a	a 指向的对象。
-------	---	----	----------

► 双目运算符的 ***** 是用于乘法计算的算术运算符。

接下来我们继续思考赋值的情况。将指向 `sanaka` 的指针赋于 `isako`，使 `isako` 指向 `sanaka`。

这样就变成了：`isako` 喜欢 `sanaka` ♥

`isako` 移情别恋了呢！同理，如果将指向其他对象的指针赋于指针变量，那么该指针变量就会指向这些对象。

`isako = &sanaka;`

我们再来看另一种赋值形式。我们知道当 `*hiroko` 指向 `masaki` 时，`*hiroko` 就是 `masaki` 的别名。那么，将 180 赋值于 `*hiroko` 就等同于将 180 赋值于 `masaki`。

`*hiriko = 180;`

因此，程序的运行结果如下。

```
printf("伊沙子喜欢的人的身高: %d\n", *isako);
printf("洋子喜欢的人的身高: %d\n", *hiroko);

/* 中略 */

printf("佐藤的身高: %d\n", sato);
printf("佐中的身高: %d\n", sanaka);
printf("平木的身高: %d\n", hiraki);
printf("真崎的身高: %d\n", masaki);
printf("伊沙子喜欢的人的身高: %d\n", *isako);
printf("洋子喜欢的人的身高: %d\n", *hiroko);
```

运行结果

伊沙子喜欢的人的身高: 178
洋子喜欢的人的身高: 179

佐藤的身高: 178
佐中的身高: 175
平木的身高: 165
真崎的身高: 180
伊沙子喜欢的人的身高: 175
洋子喜欢的人的身高: 180

① 指针运算符，也称间接访问运算符。

10-2 指针和函数

作为函数参数的指针

洋子对于恋人的要求比较高，不过她具有超能力，所以如果恋人的身高低于 180cm，她就能将他变为 180cm。现在我们就用函数来实现洋子的超能力。

当然，右图所示的程序是实现不了的。原因正如本章开头所述，函数形参无论怎么修改，都只是临时性的复制，并不会反映到主调函数的实参中。

如果不能直接修改这个值，那么就通过指针间接地修改吧。见代码清单 10-4 所示程序。

```
void hiroko(int height)
{
    if (height < 180)
        height = 180;
}
```

代码清单 10-4

```
/*
 * 函数的参数和指针
 */

#include <stdio.h>

/*--- 洋子（让身高不到 180cm 的人长到 180cm） ---*/
void hiroko(int *height)
{
    if (*height < 180)
        *height = 180;
}

int main(void)
{
    int sato = 178; /* 佐藤的身高 */
    int sanaka = 175; /* 佐中的身高 */
    int hiraki = 165; /* 平木的身高 */
    int masaki = 179; /* 真崎的身高 */

    hiroko(&masaki);

    printf("佐藤的身高: %d\n", sato);
    printf("佐中的身高: %d\n", sanaka);
    printf("平木的身高: %d\n", hiraki);
    printf("真崎的身高: %d\n", masaki);

    return (0);
}
```

运行结果

```
佐藤的身高: 178
佐中的身高: 175
平木的身高: 165
真崎的身高: 180
```

通过以下语句调用 *hiroko* 函数会发生什么情况呢？请看图 10-5。

```
hiroko(&masaki);
```

hiroko 函数中，形参 *height* 被声明为“指向 **int** 型变量的指针变量”。函数被调用时，将 **&masaki**（即 106 号）复制到 *height* 中，指针 *height* 便指向了 *masaki*。

由于在指针前加上指针运算符 *****，就可显示该指针指向的对象。因此 ***height** 是 *masaki* 的别名。

若 ***height** 的值小于 180，则将 180 赋值给它。对 ***height** 赋值，也就是对 *masaki* 赋值，所以即使从 *hiroko* 函数返回 **main** 函数，*masaki* 中保存的依然是 180。

```
void hiroko(int *height)
{
    if (*height < 180)
        *height = 180;
}
```

综上所述，如果要在函数中修改变量的值，就需要传入指向该变量的指针，即告诉程序：

传入的是指针哦，请对该指针指向的对象进行处理吧。

只要在被调函数里的指针前写上指针运算符 *****，就能间接地处理该指针指向的对象。这也是 ***** 运算符又称做间接访问运算符的原因。

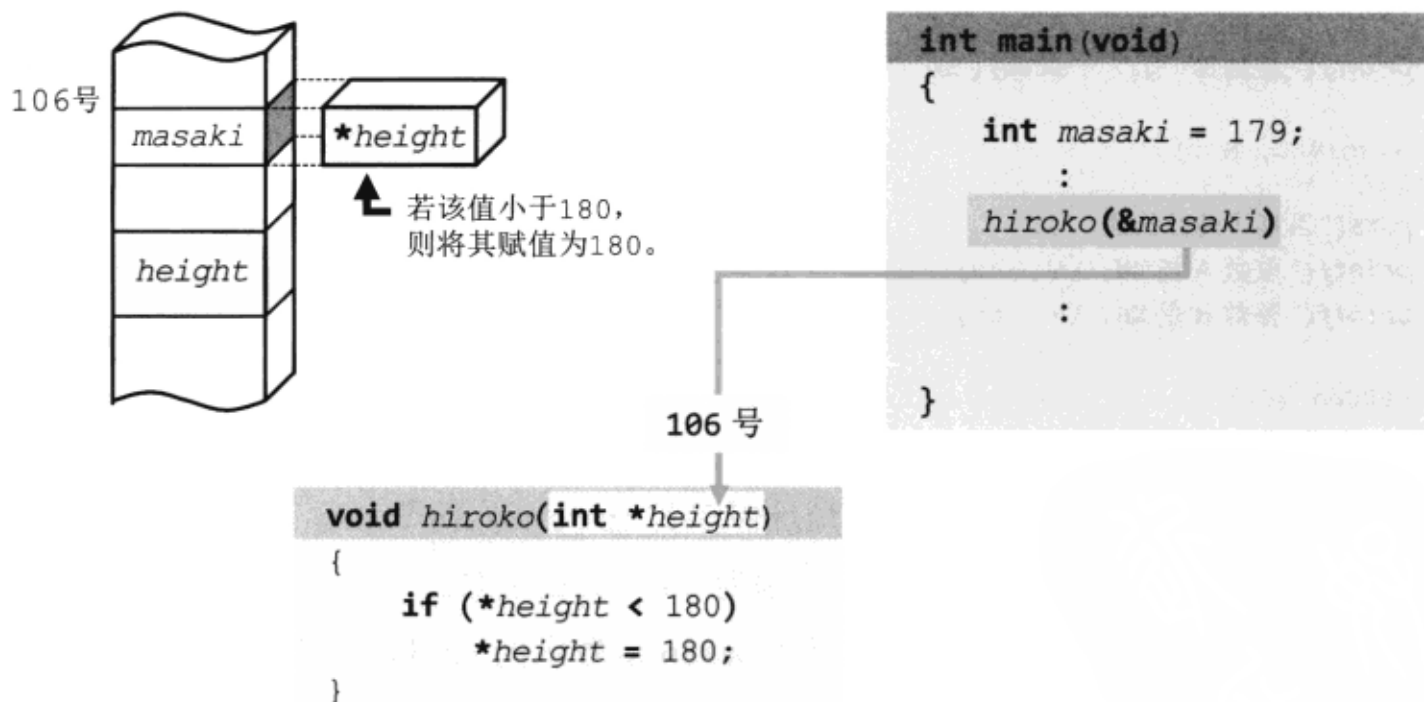


图 10-5 函数调用中指针的传递

► 图中假设 *masaki* 所在的地址为 106 号。今后若文中无特别说明，则都将在图中任意假定一个地址。

二值互换

代码清单 10-5 所示程序的功能是将两个 **int** 型对象的值互换。

代码清单 10-5

```
/*
  将两个整数值互换
*/

#include <stdio.h>

/*--- 将 nx、ny 指向的对象的值进行互换 ---*/
void swap(int *nx, int *ny)
{
    int temp = *nx;
    *nx = *ny;
    *ny = temp;
}

int main(void)
{
    int na, nb;

    puts("请输入两个整数。");
    printf("整数 A: "); scanf("%d", &na);
    printf("整数 B: "); scanf("%d", &nb);

    swap(&na, &nb);

    puts("互换了两数的值。");
    printf("整数 A 是 %d.\n", na);
    printf("整数 B 是 %d.\n", nb);

    return (0);
}
```

运行结果

请输入两个整数。
 整数 A:
 整数 B:
 互换了两数的值。
 整数 A 是 21。
 整数 B 是 57。

调用 *swap* 函数时，会将 *na*、*nb* 的地址保存到 *nx*、*ny* 这两个形参中。那么如图 10-6 所示，**nx*、**ny* 分别为 *na*、*nb* 的别名。

由于 **nx* 和 **ny* 的值进行了互换，所以 **main** 函数中的 *na* 和 *nb* 的值也实现了互换。

► 我们在第 5 章中已经学习过二值互换的步骤。

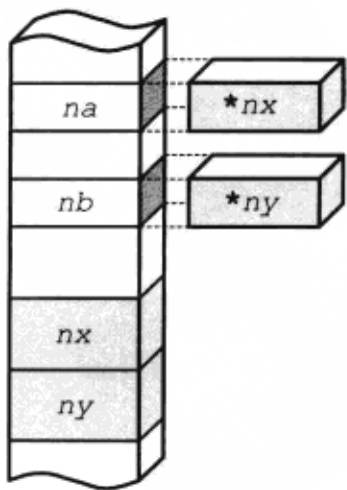


图 10-6 *swap* 函数中的指针

引用传递 (C++ 语言)

在 C 语言派生出的 C++ 语言中，可以直接对参数实体（区别于参数值）进行处理。这种结构称为引用传递（pass by reference）。

现在我们用引用传递重新改写程序，见代码清单 10-6。

代码清单 10-6

```
/*
  将两个整数值进行互换（C++ 的引用传递）
*/

#include <stdio.h>

/*--- 将 nx、ny 所指对象的值进行互换 ---*/
void swap(int& nx, int& ny)
{
    int temp = nx;
    nx = ny;
    ny = temp;
}

int main(void)
{
    int na, nb;

    puts("请输入两个整数。");
    printf("整数 A: ");    scanf("%d", &na);
    printf("整数 B: ");    scanf("%d", &nb);

    swap(na, nb);

    puts("互换了两数的值。");
    printf("整数 A 是 %d.\n", na);
    printf("整数 B 是 %d.\n", nb);

    return (0);
}
```

期望结果

请输入两个整数。
 整数 A: 21
 整数 B: 57
 互换了两数的值。
 整数 A 是 57。
 整数 B 是 21。

► 该程序在 C 语言编译器中不能编译和运行。

这样，形参 *nx*、*ny* 就分别成为了实参 *na*、*nb* 的别名。所以在 *swap* 函数中不使用指针运算符，也能直接修改主调函数中 *na*、*nb* 的值。

► 引用传递使主调函数和被调函数通过参数紧密联系在了一起，这就意味着函数的“模块独立性”降低了。所以引用传递虽然很方便，但若使用不当则会造成程序的质量下降（这就是 C 语言不支持引用传递的原因）。

计算和与差

本章开头计算两个整数和与差的程序没有运行成功，现在大家已经想到该如何修改了吧。正确程序见代码清单 10-7。

代码清单 10-7

```
/*
  计算两个整数的和与差
*/

#include <stdio.h>

/*--- 将 n1 和 n2 的和、差分别保存至 sum、diff ---*/
void sum_diff(int n1, int n2, int *sum, int *diff)
{
    *sum = n1 + n2;
    *diff = (n1 > n2) ? n1 - n2 : n2 - n1;
}

int main(void)
{
    int na, nb;
    int wa = 0, sa = 0;

    puts("请输入两个整数。");
    printf("整数A: ");    scanf("%d", &na);
    printf("整数B: ");    scanf("%d", &nb);

    sum_diff(na, nb, &wa, &sa);

    printf("两数之和是 %d。\\n 两数之差是 %d。\\n", wa, sa);

    return (0);
}
```

运行结果

请输入两个整数。
整数A:
整数B:
两数之和是 78。
两数之差是 36。

scanf 函数和指针

我们在第 1 章遇到 **scanf** 函数的时候，曾这样讲过：

但与 **printf** 函数不同的是，这里的变量名之前必须加上一个特殊符号 **&**，请大家注意。

► **&** 的具体含义，会在第 10 章中进行说明。

scanf 函数的使命是为主调函数中定义的对象保存值。倘若它接收到的纯粹是变量的“值”，是无法进行保存的。因此，**scanf** 函数接收的是指针（具有地址的“值”），由该指针所指对象保存从标准输入（一般为键盘）读到的值。

将两个值升序排列

代码清单 10-8 所示程序的功能为将两个 **int** 型对象的值升序排列。

代码清单 10-8

```
/*
  将两个整数值升序排列
*/

#include <stdio.h>

/*--- 将 nx、ny 所指对象的值进行互换 ---*/
void swap(int *nx, int *ny)
{
    int temp = *nx;
    *nx = *ny;
    *ny = temp;
}

/*--- 排列顺序为 *n1≤*n2---*/
void sort2(int *n1, int *n2)
{
    if (*n1 > *n2)
        swap(n1, n2);
}

int main(void)
{
    int na, nb;

    puts("请输入两个整数。");
    printf("整数 A: "); scanf("%d", &na);
    printf("整数 B: "); scanf("%d", &nb);

    sort2(&na, &nb);

    puts("将两数的值按升序排列。");
    printf("整数 A 是 %d.\n", na);
    printf("整数 B 是 %d.\n", nb);

    return (0);
}
```

运行结果

请输入两个整数。
 整数 A: 57
 整数 B: 21
 将两数的值按升序排列。
 整数 A 是 21。
 整数 B 是 57。

运行结果

请输入两个整数。
 整数 A: 37
 整数 B: 49
 将两数的值按升序排列。
 整数 A 是 37。
 整数 B 是 49。

`sort2` 函数调用 `swap` 函数，使 `n1` 指向的变量总是小于等于 `n2` 指向的变量。`n1` 和 `n2` 这两个指针指向的是保存待排列数值的变量，因此将它们直接传入 `swap` 函数。请注意调用 `swap` 函数的语句不能改为：

```
swap(&n1, &n2);
```

- `&n1` 的数据类型为“指向 `int` 型的指针的指针”。这个概念已超出本书的范围，在此就不详细展开了。

指针的类型

代码清单 10-9 的程序中，`swap` 函数的功能是将两个 `int` 型整数进行互换，而传入的却是指向 `double` 型变量的指针。

编译程序时，（多数编译器）会显示警告信息。而且果不其然，运行结果中显示的也不是正常的值。

► 有些编译器可能会顺利运行。

根据图 10-7 可知，指针 `nx` 指向了 `double` 型变量 `dx`，但是 `int` 型的 `*nx` 却不能等同于 `double` 型变量 `dx`。

► 当然，即使在某些情况下 `sizeof(int)` 和 `sizeof(double)` 长度碰巧相同，但如果对于位的解析不同，数据类型也会有所区别。

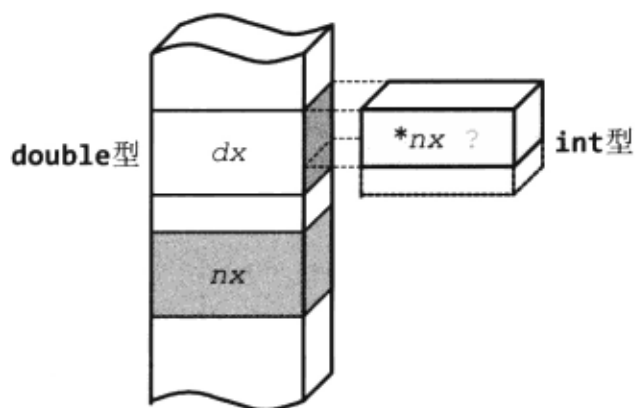


图 10-7 指针的类型和指针运算符

指针不只是表示指向“〇〇号”，更确切地说是指向“以〇〇号为首地址的 `××` 型对象”。因此，指向 `int` 型对象时必须使用 `int` 型指针，指向 `double` 型对象时必须使用 `double` 型指针。

专题 10-1 取不到地址的对象

对于使用 `register` 关键字声明的寄存器对象，不能加上取址运算符 `&`。因此，下述程序在编译时会报错。

```
#include <stdio.h>

int main(void)
{
    register int x;

    printf("%p\n", &x);

    return (0);
}
```


代码清单 10-9

```

/*
 将两个实数值进行互换（误例）
*/

#include <stdio.h>

/*--- 将 nx、ny 所指对象进行互换 ---*/
void swap(int *nx, int *ny)
{
    int temp = *nx;
    *nx = *ny;
    *ny = temp;
}

int main(void)
{
    double dx, dy;

    puts("请输入两个实数。");
    printf("实数 X: "); scanf("%lf", &dx);
    printf("实数 Y: "); scanf("%lf", &dy);

    swap(&dx, &dy);

    puts("互换了两数的值。");
    printf("实数 A 是 %f.\n", dx);
    printf("实数 B 是 %f.\n", dy);

    return (0);
}

```

运行结果

请输入两个实数。
 实数 X: 53.5
 实数 Y: 21.68
 互换了两数的值。
 实数 A 是 9980.450456。
 实数 B 是 50.568782。

► 运行结果因编译器和运行环境或有不同。

标量型

可以将表示地址编号的指针视为一种数量。第 7 章介绍的数值型和本章介绍的指针型统称标量型 (scalar type)。

● 练习 10-1

编写如下函数，求出并设置 y 年 m 月 d 日的前一天或后一天的日期（能正确判断闰年）。

```

void yesterday(int *y, int *m, int *d) { /* ... */ }
void tomorrow(int *y, int *m, int *d) { /* ... */ }

```

● 练习 10-2

编写如下函数，将三个 `int` 型整数按升序排列。

```

void sort3(int *n1, int *n2, int *n3) { /* ... */ }

```

10-3 指针和数组

指针和数组

请看代码清单 10-10 的程序。指针 *ptr* 的初始值为数组 *vc* 的第一个元素 *vc*[0]。

代码清单 10-10

```
/*
 数组和指针
*/

#include <stdio.h>

int main(void)
{
    int i;
    int vc[5] = {10, 20, 30, 40, 50};
    int *ptr = &vc[0];

    for (i = 0; i < 5; i++)
        printf("vc[%d] = %d    ptr[%d] = %d    *(ptr + %d) = %d\n",
              i, vc[i], i, ptr[i], i, *(ptr + i));

    return (0);
}
```

运行结果

vc[0] = 10	ptr[0] = 10	*(ptr + 0) = 10
vc[1] = 20	ptr[1] = 20	*(ptr + 1) = 20
vc[2] = 30	ptr[2] = 30	*(ptr + 2) = 30
vc[3] = 40	ptr[3] = 40	*(ptr + 3) = 40
vc[4] = 50	ptr[4] = 50	*(ptr + 4) = 50

如图 10-8 所示，*ptr* 指向了 *vc*[0]，因此 **ptr* 就是 *vc*[0] 的别名。

► 图 10-8 显示的是 **sizeof(int)** 为 2 的情况。当 *vc*[0] 的首地址为 500 号时，下一个元素 *vc*[1] 的保存位置就是 502 号。因此，*ptr* + 1 指向的不是 500 号加 1 所得的 501 号，而是加上 **sizeof(int)** 所得的 502 号。

C 语言中有下述规则，指针 *ptr* 加 *i* 或减 *i* (*i* 为整数) 的表达式也是指针。

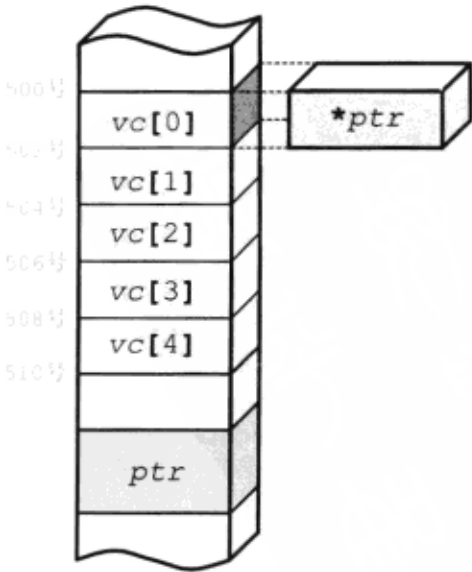


图 10-8 指向数组第一个元素的指针

$ptr + i$ 为指向 ptr 所指对象后第 i 个元素的指针。

$ptr - i$ 为指向 ptr 所指对象前第 i 个元素的指针。

因此, $ptr + 2$ 指向的是 $vc[2]$, $ptr + 4$ 指向的是 $vc[4]$ 。

由于指针 $ptr + 2$ 指向了 $vc[2]$, 所以加上指针运算符 $*$ 的表达式 $*(ptr + 2)$ 就是 $vc[2]$ 的别名。

那么, 我们就要记住当 ptr 是指针, 且 i 为整数时以下规则成立。

$*(ptr + i)$ 也可以写作 $ptr[i]$ 。

根据这个规则, 当指针 $ptr + 2$ 指向 $vc[2]$ 时, $vc[2]$ 的别名 $*(ptr + 2)$ 可以写作 $ptr[2]$ 。

$vc[2] \quad *(ptr + 2) \quad ptr[2]$

也就是说这三者都能表示数组 vc 的第 3 个元素。

如图 10-9 所示, 如果对指针使用 $[]$ 运算符, 就能像数组那样进行操作。

► 本书讲解了当指针 ptr 指向数组 vc 的首地址时, 以下三个表达式都表示数组 vc 的第 3 个元素。

$vc[2] \quad *(ptr + 2) \quad ptr[2]$

双目运算符 $+$ 没有规定操作数的顺序, 所以也可以将表达式 $ptr + 2$ 写作 $2 + ptr$ 。下标运算符 $[]$ 也同样对操作数的顺序没有限制。还有, 不带 $[]$ 的数组名会被视为指向该数组的第一个元素的指针。综上所述, 下述 8 个表达式都表示数组 vc 的第 3 个元素。

$vc[2] \quad 2[vc] \quad *(vc + 2) \quad *(2 + vc) \quad *(ptr + 2) \quad *(2 + ptr) \quad ptr[2] \quad 2[ptr]$

当然, 最好还是不使用容易让人产生误解的写法。

这个规则对于指针和数组来说都非常重要, 请务必牢记于心。

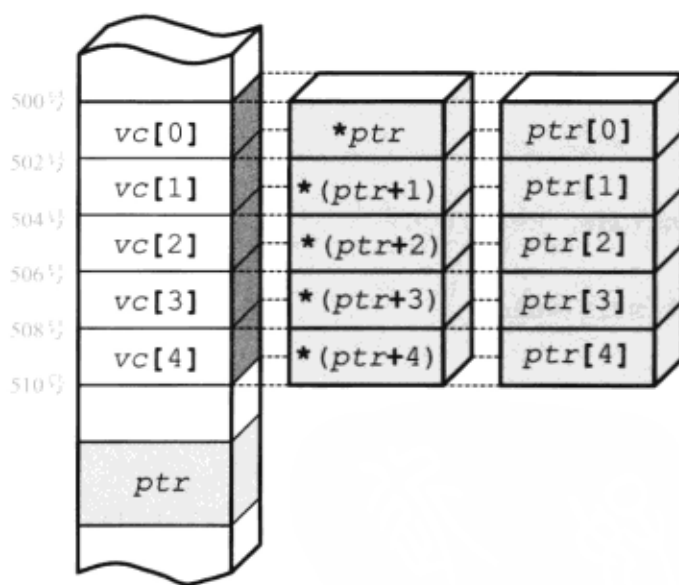


图 10-9 数组和指针

■ 注意 ■

程序中不带下标运算符 `[]` 而单独出现的数组名，会被视为指向该数组的第一个元素的指针。

例如，假设 `vc` 为数组，则不带 `[]` 单独出现的数组名 `vc`，就是指向该数组第一个元素 `vc[0]` 的指针，即表示 `&vc[0]`。

验证上述内容的程序见代码清单 10-11。从程序的运行结果可知 `vc` 和 `&vc[0]` 的值是相同的。

► 原则上，程序中不带下标运算符 `[]` 而单独出现的数组名，是指向该数组第一个元素的指针，但也有例外情况，具体如下。

(1) 数组名为 `sizeof` 运算符的操作数时

当 `sizeof` 运算符的操作数为数组名时，会生成该数组的长度。

(2) 数组名为 `&` 运算符的操作数时

当取址运算符 `&` 的操作数为数组名时，会生成指向该数组的指针，而不是指向数组第一个元素的指针。

代码清单 10-11

```
/*
 * 确认不带下标运算符的数组名的值（指向第一个元素的指针）
 */

#include <stdio.h>

int main(void)
{
    int vc[3];

    printf("vc          : %p\n", vc);
    printf("vc[0] 的地址: %p\n", &vc[0]);
    printf("vc[1] 的地址: %p\n", &vc[1]);
    printf("vc[2] 的地址: %p\n", &vc[2]);

    return (0);
}
```

运行结果

```
vc          : FFB0
vc[0] 的地址: FFB0
vc[1] 的地址: FFB2
vc[2] 的地址: FFB4
```

► 运行结果因编译器和运行环境或有不同。

另外，指针与指针之间也可以用等于运算符和关系运算符进行比较，还可以用双目运算符 `-` 进行减法运算。我们来看代码清单 10-12 所示的程序。

代码清单 10-12

```

/*
  指针的比较
*/

#include <stdio.h>

int main(void)
{
    int    vc[3];
    int    *ptr = vc;

    printf("vc    == ptr    : %d\n", vc    == ptr);
    printf("&vc[1] <= &vc[1] : %d\n", &vc[1] <= &vc[1]);
    printf("&vc[1] <  &vc[2] : %d\n", &vc[1] <  &vc[2]);
    printf("&vc[2] -  &vc[0] : %d\n", &vc[2] -  &vc[0]);

    return (0);
}

```

运行结果

```

vc    == ptr    : 1
&vc[1] <= &vc[1] : 1
&vc[1] <  &vc[2] : 1
&vc[2] -  &vc[0] : 2

```

指针 *ptr* 的初始值为指向数组 *vc* 的第一个元素，所以等价表达式 *vc == ptr* 成立，表达式的值为 1。

接下来是两个比较运算。比较的规则是：指向较后面的元素的指针“大”，而指向较前面的元素的指针“小”。

最后的减法运算则是计算两个指针之间间隔几个元素。*vc[0]* 和 *vc[2]* 的间隔为 2，因此 *&vc[2] - &vc[0]* 的值为 2。

专题 10-2 指针可指向的范围

请看前面的图 10-9。数组 *vc* 的最后一个元素为 *vc[4]*，我们并不知道该元素后面（图的下方）的内存空间保存了什么。所以，我们不能轻易对这样的内存空间进行读写操作。

表达式 *ptr + i*、*ptr - i* 分别指向了指针 *ptr* 所指元素后面、前面的第 *i* 个对象，其实这里的 *i* 的大小是有一定限制的。

具体而言，*ptr + i*、*ptr - i* 可以正确表示的范围是，指向数组第一个元素的指针 *&vc[0]* 到指向数组末尾元素的下一个元素的指针 *&vc[5]*。

也就是说，当 *ptr* 指向 *vc[0]* 时，正确的指针是 *ptr + 0* 至 *ptr + 5*。而 *ptr - 1*、*ptr + 11* 等表达式就无法确保正确地指向数组 *vc* 前后的内存空间。

因此，如果 *ptr* 指向了 *vc[2]*，那么正确的指针范围就是 *ptr - 2* 到 *ptr + 3*。

而所谓的可以指向数组末尾的下一个元素，则是对于使用哨兵查找等算法的程序（第 6 章）而言的。

数组的传递

请看代码清单 10-13 所示的程序。`int_set` 函数会从数组 `vc` 的第一个元素开始依次将变量 `val` 赋给数组的 `no` 个元素。

代码清单 10-13

```
/*
 * 数组的传递
 */
#include <stdio.h>

void int_set(int vc[], int no, int val)
{
    int i;

    for (i = 0; i < no; i++)
        vc[i] = val;
}

int main(void)
{
    int i;
    int ary[] = {1, 2, 3, 4, 5};

    int_set(ary, 5, 0);

    for (i = 0; i < 5; i++)
        printf("ary[%d] = %d\n", i, ary[i]);

    return (0);
}
```

运行结果

```
ary[0] = 0
ary[1] = 0
ary[2] = 0
ary[3] = 0
ary[4] = 0
```

我们先来看 `int_set` 函数的开头：

```
void int_set(int vc[], int no, int val)
```

其中“`int vc[]`”表示接收的参数数据类型是数组。该声明也可以写作：

```
void int_set(int *vc, int no, int val)
```

在第 6 章中曾经提到过，这个知识点比较难，需要指针知识方能更好地理解，所以放到本章进行讲解。

接着再来看调用 `int_set` 函数的地方。

```
int_set(ary, 5, 0);
```

我们已经知道单独出现的数组名是指向第一个元素的指针，所以第一个实参 `ary` 就是指向数组 `ary` 的第一个元素 `ary[0]` 的指针，即 `&ary[0]`。

那么调用 `int_set` 函数时，形参 `vc` 保存的就是 `&ary[0]` 的值。因而指针 `vc` 亦指向数组

`ary` 的第一个元素 `ary[0]`。我们可以通过图 10-10 了解到它们在内存中的表现。

因为指针 `vc` 指向数组 `ary` 的第一个元素，所以下面的图式成立。

```
ary[0] = *vc      = vc[0]
ary[1] = *(vc + 1) = vc[1]
      :          :
ary[4] = *(vc + 4) = vc[4]
```

由此可见，在 `int_set` 函数中，对指针 `vc` 使用下标运算符 `[]`，就能将 `vc` 当作数组进行处理。

■ 注意 ■

在函数间传递数组，是以指向第一个元素的指针形式进行的。在被调函数中，对该指针使用下标运算符 `[]`，就能像操作数组那样对它进行处理。

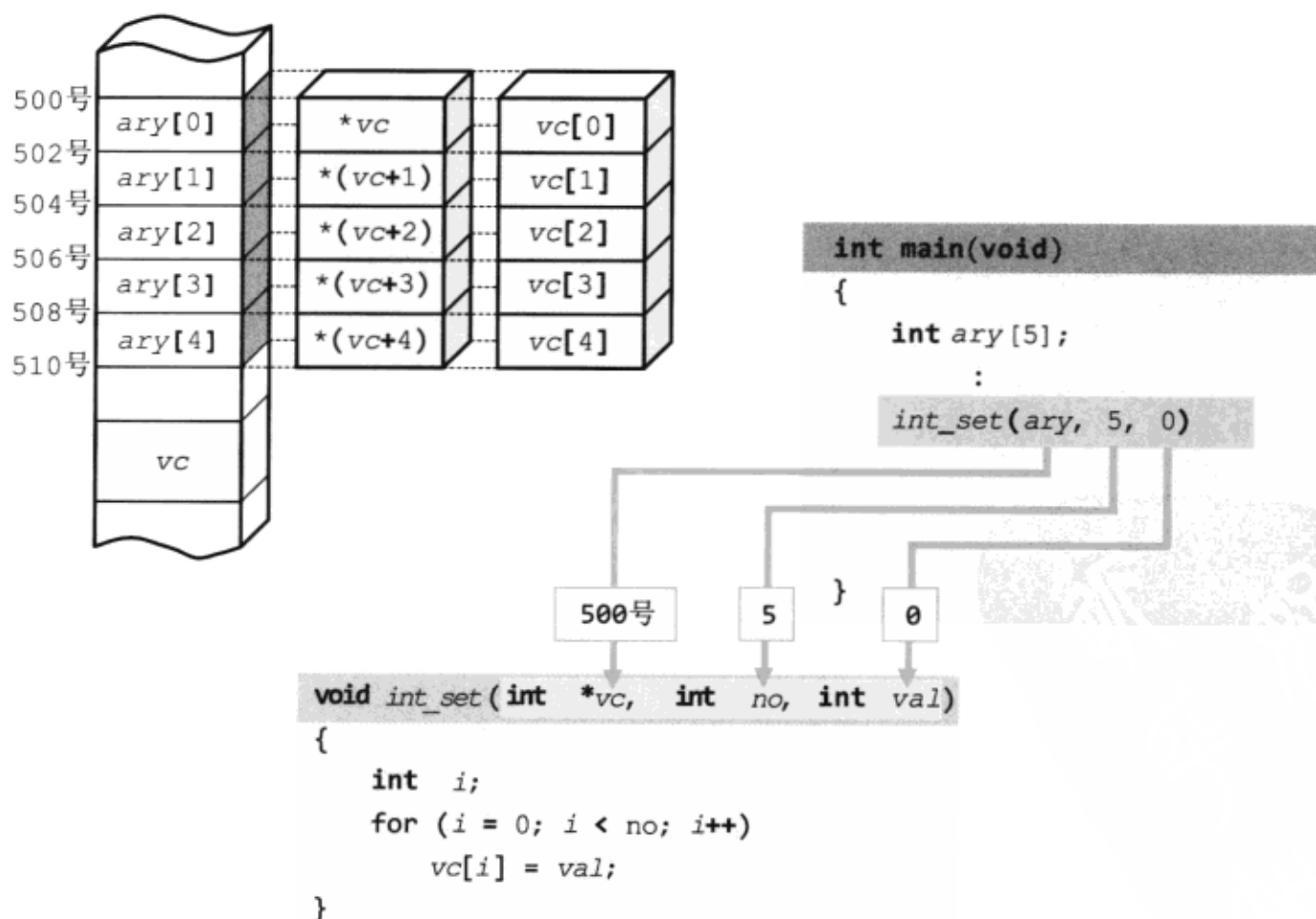


图 10-10 数组的传递



第 11 章

字符串和指针

我们在第 9 章和第 10 章分别学习了字符串和指针。这两者具有非常密切的关系。如果把这个关系理解透彻，便可游刃有余地处理字符串了。

本章就来学习字符串和指针的基本知识。



11-1 字符串和指针

字符串和指针

从上一章的学习中，我们了解了数组和指针具有相似性。以字符数组形式表示的字符串，也可以借由指针来获得更灵活多样的处理。代码清单 11-1 的程序中，两个字符串的实现方式不同。

代码清单 11-1

```
/*
 * 用数组实现的字符串和用指针实现的字符串
 */
#include <stdio.h>

int main(void)
{
    char str[] = "ABC"; /* 用数组实现的字符串 */
    char *ptr = "123"; /* 用指针实现的字符串 */

    printf("str = \"%s\"\n", str);
    printf("ptr = \"%s\"\n", ptr);

    return (0);
}
```

运行结果

```
str = "ABC"
ptr = "123"
```

`ptr` 是指向 `char` 型变量的指针变量，它的初始值为字符串字面量 `"123"`。这里请记住下述要点。

■ 注意 ■

在操作字符串字面量时，其数据类型是指向 `char` 型变量的指针，指向的是该字符串字面量的第一个字符。

所以 `ptr` 被初始化为指向保存在内存中的字符串字面量 `"123"` 的第一个字符 `'1'` 的指针。

如果字符串字面量 `"123"` 的首地址为 300 号，那么 `ptr` 的初始值就是 300 号。通过比较图 11-1 和图 11-2 可以发现，这两个字符串的实现方式是完全不同的。在本书中，将 `str` 那样的字符串称为：用数组实现的字符串。而将 `ptr` 那样的字符串称为：用指针实现的字符串。

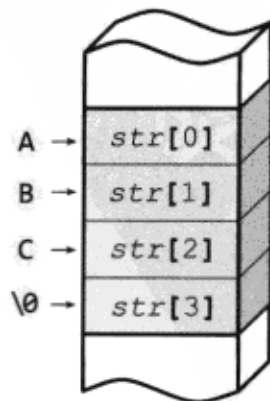


图 11-1 用字符数组实现的字符串

数组和指针的相同点

我们通过代码清单 11-2 中的程序来看看用数组实现的字符串和用指针实现的字符串有何相同之处。

代码清单 11-2

```
/*
用数组实现的字符串和用指针实现的字符串的相同点
*/

#include <stdio.h>

int main(void)
{
    int    i;
    char   str[] = "ABC";
    char   *ptr   = "123";

    for (i = 0; str[i]; i++)
        putchar(str[i]);          /* str[i] 是第一个字符之后的第 i 个元素 */
    putchar('\n');

    for (i = 0; ptr[i]; i++)
        putchar(ptr[i]);          /* ptr[i] 是第一个字符之后的第 i 个元素 */
    putchar('\n');

    printf("str = \"%s\"\n", str); /* str 是指向第 1 个字符的指针 */
    printf("ptr = \"%s\"\n", ptr); /* ptr 是指向第 1 个字符的指针 */

    return (0);
}
```

运行结果

```
ABC
123
str = "ABC"
ptr = "123"
```

请注意上表中蓝色底纹的代码。我们讲过不带下标运算符 `[]` 单独出现的数组名，就是指向该数组第一个元素的指针（10-3 节），所以 `str` 就是指向第一个字符 `str[0]`（即 'A'）的指针。当然，原本就是指针的 `ptr` 则指向字符串字面量 "123" 的第一个字符 '1'。

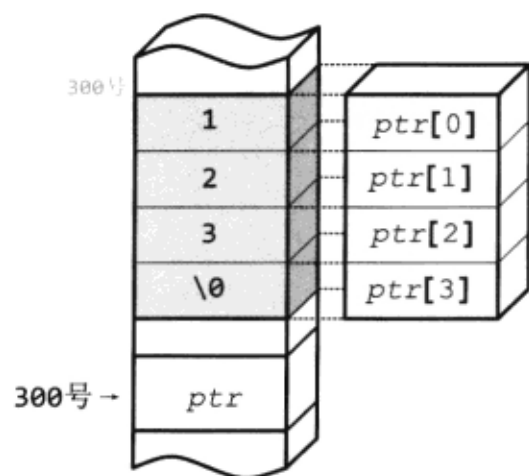


图 11-2 用指针实现的字符串

传入 `printf` 函数的 `str` 和 `ptr`，一个是数组，一个是指针。这两者有着本质区别，但它们都指向字符串的第一个字符，这一点是相同的。

如果将下标运算符 `[]` 用于指针，就能像数组那样进行操作（10-3 节）。

例如，`ptr[1]` 是第二个字符 '2' 的别名。

这也是它们的相同点，即可以使用下标运算符表示字符串中的任意字符。

数组和指针的不同点

我们再通过代码清单 11-3 中的程序来观察一下用数组实现的字符串和用指针实现的字符串有何本质区别。

代码清单 11-3

```
/*
  字符串赋值 (?)
*/

#include <stdio.h>

int main(void)
{
    char    str[] = "ABC";
    char    *ptr  = "123";

    str = "DEF";      /* 错误：不能这样赋值 */
    ptr = "456";      /* 正确：指向另一个字符串字面量 */

    printf("str = \"%s\"\n", str);
    printf("ptr = \"%s\"\n", ptr);

    return (0);
}
```

► 该程序无法编译和运行。

对于用数组实现的 *str*，不能再赋值为其他字符串，这在第 9 章中已作说明。

不过对于指针是可以重新赋值的。在这个程序中，

原来：

ptr 喜欢 "123" ♥

后来见异思迁，变为：

ptr 喜欢 "456" ♥

当然，这并不会复制整个字符串的内容。如图 11-3 所示，只是重新指向了另一个字符串（的第一个字符）。

```
char    str[] = "ABC";
str = "DEF";    /* 错误 */
```

```
char    *ptr = "123";
ptr = "456";    /* 正确 */
```

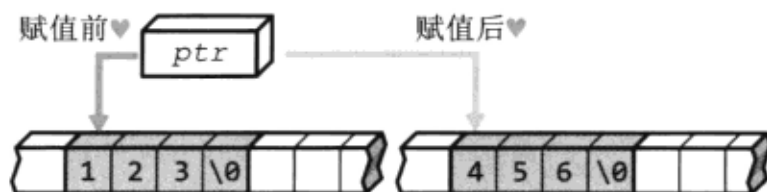


图 11-3 对指针赋值

代码清单 11-4 所示的程序，只是在代码清单 11-1 的基础上多加了一个字符。

代码清单 11-4

```
/*
用数组实现的字符串和用指针实现的字符串（其二）
*/
#include <stdio.h>

int main(void)
{
    char str[6] = "ABC";          /* 用数组实现的字符串 */
    char *ptr = "123";           /* 用指针实现的字符串 */

    printf("str = \"%s\"\n", str);
    printf("ptr = \"%s\"\n", ptr);

    return (0);
}
```

运行结果

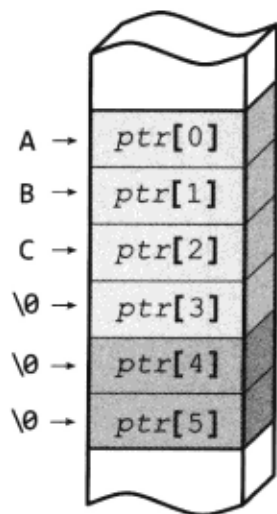
```
str = "ABC"
ptr = "123"
```

这里只是稍稍增加了数组的元素数，运行结果没有变化。

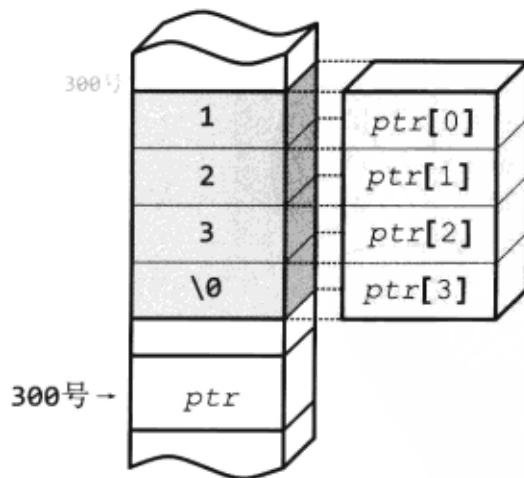
不过这样一写，用数组实现的字符串和用指针实现的字符串之间的区别就显而易见了。

数组的元素是在连续的内存单元中保存的，所以数组 *str* 的 6 个元素存在于连续的内存单元中。如果第一个 null 字符之后的部分（图 11-4a）没有其他用途，那么它们就成了多余的内存空间。

再看用指针 *ptr* 实现的字符串，其初始值为指向字符串字面量 "123"（的第一个字符）的指针，根据图示可知它不会占用多余的空间。当然，字符串字面量本身以及指针都是需要占用一定的内存空间的（图 11-4b）。



(a) 用数组实现的字符串



(b) 用指针实现的字符串

图 11-4 字符串的实现

字符串数组

我们在第9章学习了“用数组实现的字符串”的数组，本节继续学习“用指针实现的字符串”的数组。

代码清单 11-5 所示的程序分别展示了用这两种方式实现的字符串数组。

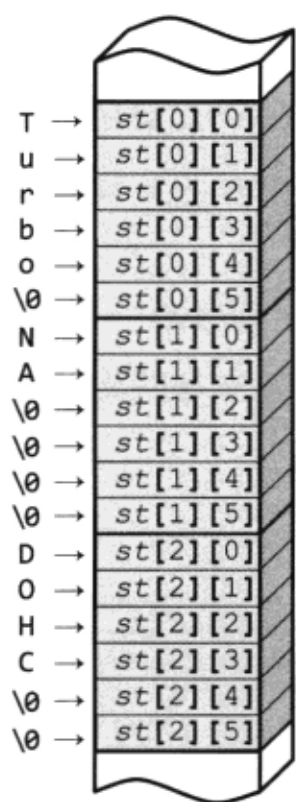
二维数组 *st* 的初始值为 "Turbo"、"NA"、"DOHC" 这三个字符串。其中最长的字符串 "Turbo" 包括 null 在内共有 6 个字符，所以该二维数组的大小为 3×6。

而数组 *pt* 为

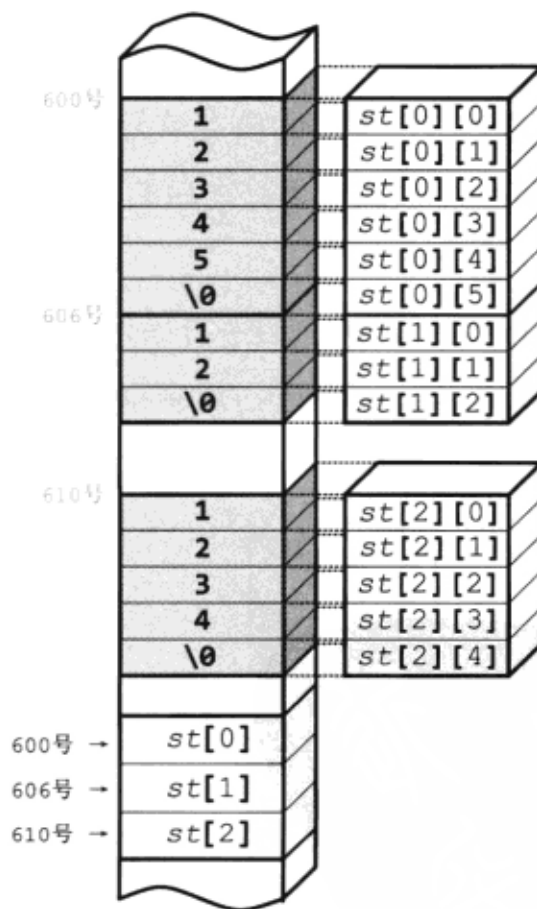
具有三个数据类型为“指向 **char** 型变量的指针型”的元素的数组。

数组元素 *pt*[0]、*pt*[1]、*pt*[2] 都是指向 **char** 型变量的指针。程序中给定的初始值为 "12345"、"12"、"1234"，因此它们分别指向这些字符串（的第一个字符）。

请看图 11-5 比较两者的区别。



(a) 用数组实现的字符串的数组



(b) 用指针实现的字符串的数组

图 11-5 字符串数组

代码清单 11-5

```
/*
用“数组实现的字符串”的数组和“用指针实现的字符串”的数组
*/

#include <stdio.h>

int main(void)
{
    int    i;
    char  st[3][6] = {"Turbo", "NA", "DOHC"};
    char  *pt[3]   = {"12345", "12", "1234"};

    for (i = 0; i < 3; i++)
        printf("st[%d] = \"%s\\n\", i, st[i]);

    for (i = 0; i < 3; i++)
        printf("pt[%d] = \"%s\\n\", i, pt[i]);

    return (0);
}
```

运行结果

```
st[0] = "Turbo"
st[1] = "NA"
st[2] = "DOHC"
pt[0] = "12345"
pt[1] = "12"
pt[2] = "1234"
```

下面简单归纳一下两者在实现上的异同点。

相同点

数组和指针都能使用下标运算符 []，两者表示字符串中各个字符的方法相同。

例如，`st[0][4]` 表示 'o'，`pt[0][4]` 表示 '5'。

不同点

数组的元素是在连续的内存单元中保存的，所以二维数组 `st` 的各个元素也保存在连续的内存单元中。因此，占用的空间为 $3 \times 6 = 18$ 个字符，"NA" 之后的 3 个字符空间为多余的。

而指针数组的初始值为指向 "12345"、"12"、"1234" 这 3 个字符串（的第一个字符）的指针。它和二维数组不同，不会有多余空间。不过除了 3 个字符串字面量以外，还需存放 3 个指针。

► 图 11-5(b) 中，"12" 和 "1234" 并不是相邻的。这说明 3 个字符串字面量不一定是在连续的内存单元中保存的。因此我们在编写程序的时候，不能想当然地认为保存 "12345" 的内存空间后面一定紧接着保存了 "12"。否则，在有些编译器和运行环境中，程序将不能正确运行。

11-2 通过指针操作字符串

字符串和指针

我们在上一章提到了数组和指针有着密切的关系。而用数组实现的字符串也与指针有着密不可分的因缘。

判断字符串长度

9-3 节中求字符串长度的函数，也可以改用指针的方式实现。请看代码清单 11-6 所示的程序。

代码清单 11-6

```
/*
 * 判断字符串的长度（指针版）
 */

#include <stdio.h>

/*--- 返回字符串 s 的长度 ---*/
size_t str_length(const char *s)
{
    size_t len = 0;

    while (*s++)
        len++;
    return (len);
}

int main(void)
{
    char st[100];

    printf("请输入字符串:");
    scanf("%s", st);

    printf("字符串 %s 的长度为 %u。 \n", st, (unsigned)str_length(st));

    return (0);
}
```

运行结果

请输入字符串: XYZ
字符串 XYZ 的长度为 3。

const

在指针声明中，若在数据类型前加上 **const** 修饰符，该指针指向的值就不能被修改了。

如右图所示，想通过 s 来重新赋值的程序是不能通过并运行的。

- ▶ 此处的变量声明，与下述语句中使用 $[]$ 进行的声明（第6章）是一样的。

```
void f(const char s[]) { /* ... */ }
```

在形参的声明中加上 **const** 后，`str_length` 函数的使用者就不必担心“传递的指针会被用来更改值”了。

```
void func(const char *s)
{
    s[1]    = '5'; /* 错误 */
    *(s + 2) = '6'; /* 错误 */
}
```

使用指针进行遍历

我们来看看 `str_length` 函数的内部结构。首先观察 **while** 语句的循环控制表达式：

```
*s++
```

由于后置自增运算符 **++** 会在判断表达式之后进行自增处理。因此判断该循环控制表达式得到的就是 $*s$ 的值，即 s 指向的字符（也就是自增处理前的字符）。

只要这个值不为 0（不为 null），**while** 语句就会不断循环下去。当然，无论这个判断的结果是什么，每次循环 s 都会自增。

在将 **++** 运算符用于指针时，大家一定要记住下面的这句话。

■ 注意 ■

指针自增，表示指向下一个对象。

换句话说， $s++$ 等价于以下赋值表达式：

```
s = s + 1
```

意为将 s 重新赋值为指向 s 的下一个元素的指针 $s + 1$ 。由此可知，更新后的 s 指向了下一个字符。

在循环体内部，会执行下述语句：

```
len++;
```

由此可知，如图 11-6 所示，每当 s 自增， len 也随之自增。

而当 s 指向的字符为 null 时，**while** 语句就结束循环。

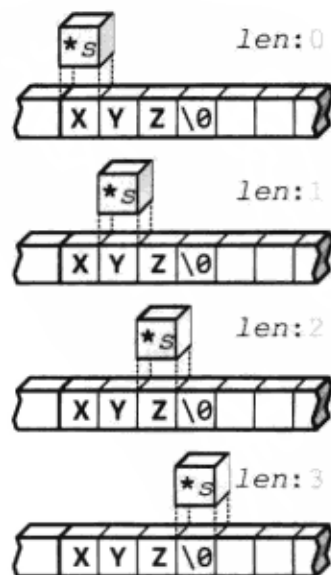


图 11-6 求字符串的长度

字符串的复制

代码清单 11-7 所示的是复制字符串的程序。

代码清单 11-7

```
/*
 * 复制字符串
 */

#include <stdio.h>

/*--- 将字符串 s 复制到 d ---*/
char *str_copy(char *d, const char *s)
{
    char *t = d;

    while (*d++ = *s++)
        ;
    return (t);
}

int main(void)
{
    char s1[128] = "ABCD";
    char s2[128] = "EFGH";

    printf("字符串 s1: ");    scanf("%s", s1);

    str_copy(s2, s1);

    puts("s1 复制到了 s2.");
    printf("s1 = %s\n", s1);
    printf("s2 = %s\n", s2);

    return (0);
}
```

运行结果

```
字符串 s1: 123
s1 复制到了 s2。
s1 = 123
s2 = 123
```

在 `str_copy` 函数中声明的指针 `t` 的初始值为 `d`，所以 `t` 和 `d` 都指向了目标字符串的第一个字符。此时的状态如图 11-7(a) 所示。

接下来看实现复制功能的 `while` 语句。在指针前加上指针运算符 `*` 形成的表达式 `*d`，就是 `d` 指向的对象的别名。因此，通过赋值表达式 `*d++ = *s++`（即 `*d = *s`），`d` 指向的字符就被赋值为 `s` 指向的字符。

完成赋值后，再判断赋值表达式 `*d++ = *s++` 的值。其判断结果为赋值后左操作数的数据类型和值，那么此处可以判断当前复制的字符是否为 0，即是否为表示字符串结束的 `null` 字符。

这时判断结果不为 `'\0'`，由于自增运算符 `++` 的作用，更新后的 `d` 和 `s` 分别指向了下一个字符。此时的状态见图 11-7(b)。

以同样步骤进行循环，直至出现 `null` 字符，则完成复制图 11-7(e)。

现在我们要思考一下，为什么不对指针变量使用下标运算符，而用指针的方式编写程序

呢？这样会稍稍增加程序的难度，其好处是什么呢？

顺便说一下，如果将指针变量 d 和 s 改写成下标运算符的方式，那么 `str_copy` 函数体就将写成右图所示的形式。其中蓝字部分告诉程序：

将 d 指向的字符之后第 i 个字符赋值为
 s 指向的字符之后第 i 个字符的值。

```
unsigned i = 0;
while (d[i] = s[i])
    i++;
```

于是程序会自行进行下标运算等内部处理，无需我们程序员关心内部细节。

回过头来看此处所示的程序，每次循环 d 和 s 都会自增，并告诉程序：

将 d 指向的字符赋值为 s 指向的字符。

```
while (*d++ = *s++)
    ;
```

因此这种情况下就不会进行下标运算等处理，也不需要用于控制循环的变量 i 。

■ 注意 ■

正确、有效地运用指针，能编写出具有良好运行效率的程序。

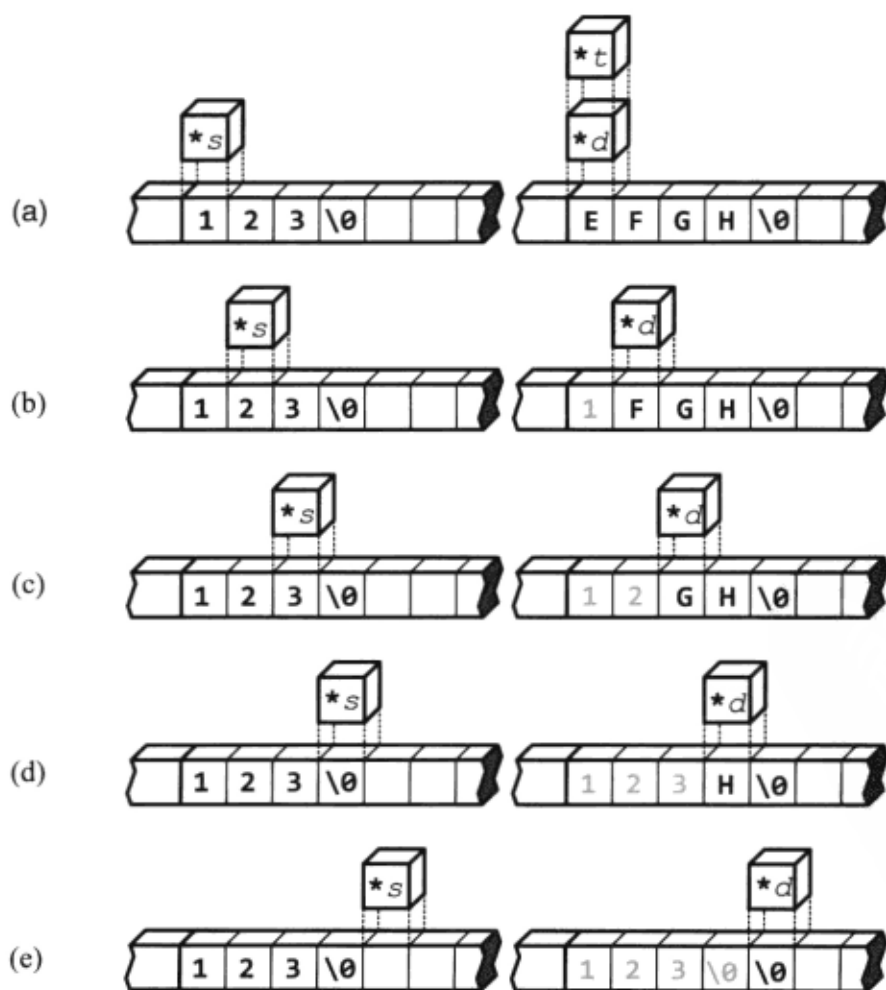


图 11-7 字符串的复制

不正确的字符串复制

请看代码清单 11-8 所示的程序。这和前面的程序大致相同。

代码清单 11-8

```
/*
 * 复制字符串（误例）
 */

#include <stdio.h>

/*--- 将字符串 s 复制到 d ---*/
char *str_copy(char *d, const char *s)
{
    char *t = d;

    while (*d++ = *s++)
        ;
    return (t);
}

int main(void)
{
    char str[128] = "ABCD";
    char *ptr      = "EFGH";

    printf("字符串 str: "); scanf("%s", str);

    str_copy(ptr, str);

    puts("将 str 复制到了 ptr。");
    printf("str = %s\n", str);
    printf("ptr = %s\n", ptr);

    return (0);
}
```

期望结果

字符串 str: 1234567

将 str 复制到了 ptr。

str = 1234567

ptr = 1234567

► 该程序不能保证运行正确。

这个程序犯了以下两个错误。

■ 改写了字符串字面量

这个程序改写了指针 *ptr* 指向的字符串字面量的内容。但是，是否可以改写字符串字面量中的字符，是取决于编译器的。在不支持改写字符串字面量的编译器中，该程序不能正确运行。

■ 可能会写入非空的内存空间。

指针 *ptr* 指向了字符串字面量 "EFGH" 的第一个字符，该字符串包括 `null` 在内长度为 5 位。

如图 11-8 所示向该内存空间复制包括 `null` 在内的 8 个字符 "1234567"。但图中蓝色底纹部分并不能保证是未使用的空内存空间。在该内存空间中可能保存着其他变量，甚至是系统的关键信息。

所以，这样复制的话可能会破坏其他变量的值，甚至可能导致程序运行异常。

■ 注意 ■

不要改写字符串字面量，也不要对超过字符串字面量的内存空间进行写入操作。

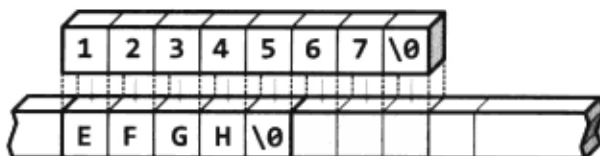


图 11-8 不正确的写入操作

返回指针的函数

再来看代码清单 11-7 中的正确程序。

`str_copy` 函数的返回类型为指向 **char** 型变量的指针型，只要是用到这种数据类型的地方都能调用该函数。

函数的返回值是指针 *t*，它复制于传入的形参 *d*。这就意味着函数返回的是“指向复制后的字符串的第一个字符的指针”。

```
str_copy(s2, s1);
```

```
printf("s2 = %s\n", s2);
```

所以，上面内容也可以简写为：

```
printf("s2 = %s\n", str_copy(s2, s1));
```

首先将字符串 *s1* 复制到字符串 *s2*，再将复制后的 *s2* 显示出来。

► 传入 `printf` 函数的正是“指向复制后的字符串的第一个字符的指针”。

11-3 字符串处理库函数

字符串处理函数

C 语言标准函数库提供了求字符串长度、复制字符串等函数。我们选取其中一些常用函数进行说明并给出应用示例。

strlen

头文件 `#include <string.h>`

原型 `size_t strlen(const char *s)`

说明 求出 *s* 指向的字符串的长度（不包括 null 字符）。

返回值 返回 *s* 指向的字符串的长度。

代码清单 11-9

```
/*--- 返回字符串 s 的长度 ---*/
size_t strlen(const char *s)
{
    size_t len = 0;

    while (*s++)
        len++;
    return (len);
}
```

● 练习 11-1

不使用下标运算符，写出与代码清单 9-9 中的 `put_string` 函数功能相同的函数。

● 练习 11-2

编写如下函数，若字符串 *str* 中含有字符 *c*（若含有多个，以先出现的为准），则返回指向该字符的指针，否则返回 NULL。

```
char *str_chr(const char *str, int c) { /* ... */ }
```

► 与指向对象（或函数）的指针相对，我们将确实“不指向任何对象”的指针称为空指针（null pointer）以示区别。空指针常用于像练习 11-2 这种表示“处理失败”等情况。

表示空指针的宏定义是 NULL，称为空指针常量（null pointer constant）。只要在预处理命令中包含 `<stddef.h>`、`<stdio.h>`、`<stdlib.h>`、`<string.h>`、`<time.h>` 中任意一个头文件，便可使用该宏定义。

strcpy**头文件** `#include <string.h>`**原型** `char *strcpy(char *s1, const char *s2)`**说明** 将 *s2* 指向的字符串复制到 *s1* 指向的数组中。若 *s1* 和 *s2* 指向的内存空间重叠，则作未定义处理。**返回值** 返回 *s1* 的值。**strncpy****头文件** `#include <string.h>`**原型** `char *strncpy(char *s1, const char *s2, size_t n)`**说明** *s2* 指向的字符串复制到 *s1* 指向的数组中。若 *s2* 的长度大于等于 *n*，则复制到第 *n* 个字符为止。否则用 `null` 字符填充剩余部分。若 *s1* 和 *s2* 指向的内存空间重叠，则作未定义处理。**返回值** 返回 *s1* 的值。**代码清单 11-10**

```

/*--- 将字符串 s2 复制到 s1 ---*/
char *strcpy(char *s1, const char *s2)
{
    char *tmp = s1;
    while (*s1++ = *s2++)
        ;
    return (tmp);
}

/*--- 将字符串 s2 的前 n 个字符复制到 s1 ---*/
char *strncpy(char *s1, const char *s2, size_t n)
{
    char *tmp = s1;

    while (n) {
        if (!(*s1++ = *s2++)) break;    /* 遇到 '\0' 就结束循环 */
        n--;
    }
    while (n--) *s1++ = '\0';           /* 用 '\0' 填充剩余部分 */
    return (tmp);
}

```

● 练习 11-3

不使用下标运算符，编写如下函数，返回字符串 *str* 中字符 *c* 的个数(若不存在，则为 0)。

```
int str_chnum(const char *str, char c) { /* ... */ }
```

strcat

头文件 **#include <string.h>**

原型 **char *strcat(char *s1, const char *s2)**

说明 将 *s2* 指向的字符串连接到 *s1* 指向的数组末尾。若 *s1* 和 *s2* 指向的内存空间重叠，则作未定义处理。

返回值 返回 *s1* 的值。

strncat

头文件 **#include <string.h>**

原型 **char *strncat(char *s1, const char *s2, size_t n)**

说明 将 *s2* 指向的字符串连接到 *s1* 指向的数组末尾。若 *s2* 的长度大于 *n* 则截断超出部分。若 *s1* 和 *s2* 指向的内存空间重叠，则作未定义处理。

返回值 返回 *s1* 的值。

代码清单 11-11

```
/*--- 将字符串 s2 添加到 s1 之后 ---*/
char *strcat(char *s1, const char *s2)
{
    char *tmp = s1;

    while (*s1) s1++;                /* 前进到 s1 的末尾处 */
    while (*s1++ = *s2++) ;          /* 循环复制直至遇到 s2 中的 '\0' */
    return (tmp);
}

/*--- 将字符串 s2 的前 n 个字符复制到 s1 --- */
char *strncat(char *s1, const char *s2, size_t n)
{
    char *tmp = s1;

    while (*s1) s1++;                /* 前进到 s1 的末尾处 */
    while (n--)                      /* 遇到 '\0' 就结束循环 */
        if (!(*s1++ = *s2++)) break; /* 在 s1 的末尾插入 '\0' */
    *s1 = '\0';
    return (tmp);
}
```

● 练习 11-4

不使用下标运算符，写出与代码清单 9-13 中的 *str_toupper* 函数和 *str_tolower* 函数功能相同的函数。

● 练习 11-5

不使用下标运算符，编写如下函数，删除字符串 *str* 中的所有数字字符（例如，将传入的 "AB1C9" 变为 "ABC"）。

```
void del_digit(char *str) { /* ... */ }
```


strcmp**头文件** `#include <string.h>`**原型** `int strcmp(const char *s1, const char *s2)`**说明** 比较 *s1* 指向的字符串和 *s2* 指向的字符串的大小关系（从第一个字符开始逐一进行比较，当出现不同的字符时，便可确定这些字符的大小关系）。**返回值** 若 *s1* 和 *s2* 相等，则返回 0；若 *s1* 大于 *s2*，则返回正整数值；若 *s1* 小于 *s2*，则返回负整数值。**strncmp****头文件** `#include <string.h>`**原型** `int strncmp(const char *s1, const char *s2, size_t n)`**说明** 比较 *s1* 指向的字符串和 *s2* 指向的字符串的前 *n* 个字符的大小关系。**返回值** 若 *s1* 和 *s2* 相等，则返回 0；若 *s1* 大于 *s2*，则返回正整数值；若 *s1* 小于 *s2*，则返回负整数值。**代码清单 11-12**

```

/*--- 比较字符串 s1 和 s2 ---*/
int strcmp(const char *s1, const char *s2)
{
    while (*s1 == *s2) {
        if (*s1 == '\0')
            return (0);          /* 相等 */
        s1++;
        s2++;
    }
    return ((unsigned char)*s1 - (unsigned char)*s2);
}

/*--- 比较字符串 s1 和 s2 的前 n 个字符 ---*/
int strncmp(const char *s1, const char *s2, size_t n)
{
    while (n && *s1 && *s2) {
        if (*s1 != *s2)          /* 不相等 */
            return ((unsigned char)*s1 - (unsigned char)*s2);
        s1++;
        s2++;
        n--;
    }
    if (!n) return (0);          /* 相等 */
    if (*s1) return (1);         /* s1 > s2 */
    return (-1);                 /* s1 < s2 */
}

```

字符串转换函数

有时我们需要将 "123"、"51.7" 这样的字符序列从数字字符串转换为整数 123 以及浮点数 51.7。为此，C 语言标准函数库提供了字符串转换函数。下面就来看看这些函数的说明。

atoi

头文件 `#include <stdlib.h>`

原型 `int atoi(const char *nptr)`

说明 将 `nptr` 指向的字符串转换为 `int` 型表示。

返回值 返回转换后的值。结果值不能用 `int` 型表示时的处理未定义（依赖于编译器）。

atol

头文件 `#include <stdlib.h>`

原型 `long atol(const char *nptr)`

说明 将 `nptr` 指向的字符串转换为 `long` 型表示。

返回值 返回转换后的值。结果值不能用 `long` 型表示时的处理未定义（依赖于编译器）。

atof

头文件 `#include <stdlib.h>`

原型 `double atof(const char *nptr)`

说明 将 `nptr` 指向的字符串转换为 `double` 型表示。

返回值 返回转换后的值。结果值不能用 `double` 型表示时的处理未定义（依赖于编译器）。

`atoi` 函数的运行情况如代码清单 11-13 所示。

► `<stdlib.h>` 的名称源于 standard library。它不同于“字符串”相关的 `<string.h>` 以及“输入输出”相关的 `<stdio.h>` 等其他头文件，从名称上看不出 `<stdlib.h>` 与什么有关。这也是因为 `<stdlib.h>` 集中将各种函数和宏定义归类于各个头文件之后，不属于任何分类。

● 练习 11-6

编写如下函数，实现与库函数 `atoi`、`atol`、`atof` 相同的功能。

```
int    strtol(const char *nptr) { /* ... */ }
long   strtol(const char *nptr) { /* ... */ }
double strtod(const char *nptr) { /* ... */ }
```

代码清单 11-13

```
/*  
    atoi 函数的运行情况  
*/
```

```
#include <stdio.h>  
#include <stdlib.h>
```

```
int main(void)
```

```
{
```

```
    char str[] = "12345";
```

```
    printf("str      = \"%s\"\n", str);
```

```
    printf("atoi(str) = %d\n", atoi(str));
```

```
    return (0);
```

```
}
```

运行结果

```
str      = "12345"  
atoi(str) = 12345
```



第 12 章

结 构 体

我们在前面 3 章学习了有关指针和字符串的内容。

本章将学习结构体，它与指针一样都是 C 语言的难点。话虽如此，但只要理解了其必要性和本质，也就不会觉得有多难了。



12-1 结构体

排序

排序(sort)就是以一定的基准,将数据的集合按升序(从小到大)或降序(从大到小)重新排列。将5名学生的身高按升序排列的程序如代码清单12-1所示。

代码清单 12-1

```
/*
   将5名学生的“身高”按升序排列
*/

#include <stdio.h>

#define NUMBER 5      /* 学生人数 */

/* --- 交换x和y指向的整数值 --- */
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

/* --- 将数组data的前n个元素按升序排列 --- */
void sort(int data[], int n)
{
    int k = n - 1;
    while (k >= 0) {
        int i, j;
        for (i = 1, j = -1; i <= k; i++)
            if (data[i - 1] > data[i]) {
                j = i - 1;
                swap(&data[i], &data[j]);
            }
        k = j;
    }
}

int main(void)
{
    int i;
    int height[] = {178, 175, 173, 165, 179};

    sort(height, NUMBER);

    for (i = 0; i < NUMBER; i++)
        printf("%2d: %4d\n", i + 1, height[i]);

    return (0);
}
```

运行结果

```
1: 165
2: 173
3: 175
4: 178
5: 179
```

冒泡排序法

现已设计出多种排序的算法（即一系列计算步骤）。在这个程序中使用了经过少许改良的冒泡排序法（bubble sorting），它是一种简单的交换排序算法。

在保存身高的数组 *height* 中，依次保存着以下数值。

178 175 173 165 179

首先看前两个数组成的数值对 [178, 175]。由于要进行升序排列，后面的值不能小于前面的值，因此要交换这两个值。

175 ↔ 178 173 165 179

接下来第 2、第 3 个数组成的数值对 [178, 173] 也作同样交换。

175 173 ↔ 178 165 179

第 3、第 4 个数组成的数值对 [178, 165] 也一样。

175 173 165 ↔ 178 179

最后一组数值对 [178, 179] 则不需要交换。

175 173 165 178 179

在以上步骤中，最后进行交换的一组数值对是 [第 3 个数，第 4 个数]。第 4 个数以后的数值，都已排序完毕（最大的是第 5 个数，其次是第 4 个数）。

第二轮，对未排序的第 1 ~ 第 3 个数进行同样操作。

175 173 165 178 179
173 ↔ 175 165 178 179
173 165 ↔ 175 178 179

这样一来，前 3 个元素中的最大值 175 就被移到了最后（前 3 个元素中的第 3 个）。

第三轮，以同样的步骤对前 2 个元素进行排序。

173 165 175 178 179
165 ↔ 173 175 178 179

这样，第 2 个数以后的元素都按升序排列了。

165 173 175 178 179

至此，最小的元素也已经排在了最前面的位置，整个排序过程完成。

请将这个过程想象成水中的气泡，较轻的气泡会不断地往上冒（本例中，会将较大的值排在右侧，较小的值排在左侧）。该算法因此得名。

数据关联性

前面的程序运行结果不能告诉我们谁的身高是多少厘米。为此我们需要定义学生名字的数据，并让它跟随身高一起排序。程序如代码清单 12-2 所示。

学生名字由字符串数组 *name* 保存。交换身高时也必须同时交换名字，这样才能正确排序。因此，除了交换整数值的 *swap* 函数之外，还定义了用于交换字符串的 *swaps* 函数。

如此一来，便能很清楚地在运行结果中显示谁的身高是多少厘米了。

*

假设现在除了名字，还需要添加 **float** 型的体重和 **long** 型的月度奖学金数据。

那么，当然也可以像刚才的程序那样，为每个项目定义相应的数组。但这就必须再添加用于交换 **float** 型和 **long** 型数据的函数。这种“限于特定类型”的编程是否妥当呢？

专题 12-1 基本的排序法

上一页提到过排序的算法有很多种。下述程序就分别运用插入排序、选择排序和未改良的冒泡排序算法，实现了整数数组的排序（*swap* 函数，请参考前面的内容）。

```

/*--- 插入排序 ---*/
void insertion(int a[], int n)
{
    int i, j;
    for (i = 1; i < n; i++) {
        int tmp = a[i];
        for (j = i; j > 0 && a[j - 1] > tmp; j--)
            a[j] = a[j - 1];
        a[j] = tmp;
    }
}

/*--- 冒泡排序 ---*/
void bubble(int a[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++) {
        for (j = 1; j <= n - i; j++)
            if (a[j - 1] > a[j])
                swap(&a[j - 1], &a[j]);
    }
}

/*--- 选择排序 ---*/
void selection(int a[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++) {
        int min = i;
        for (j = i + 1; j < n; j++)
            if (a[j] < a[min])
                min = j;
        swap(&a[min], &a[i]);
    }
}

```


代码清单 12-2

```

/*
    对 5 名学生的“姓名和身高”按身高进行升序排列
*/

#include <stdio.h>
#include <string.h>

#define NUMBER          5          /* 学生人数 */

/*--- 交换 x 和 y 指向的整数值 ---*/
void swap(int *x, int *y)
{
    /*--- 省略 (同代码清单 12-1) ---*/
}

/*--- 将 sx 和 sy 指向的字符串进行交换 ---*/
void swaps(char sx[], char sy[])
{
    char temp[256];

    strcpy(temp, sx);
    strcpy(sx, sy);
    strcpy(sy, temp);
}

/*--- 对数组 data 和 name 的前 n 个元素进行升序排列 ---*/
void sort(int data[], char name[][20], int n)
{
    int k = n - 1;
    while (k >= 0) {
        int i, j;
        for (i = 1, j = -1; i <= k; i++)
            if (data[i - 1] > data[i]) {
                j = i - 1;
                swap(&data[i], &data[j]);
                swaps(name[i], name[j]);
            }
        k = j;
    }
}

int main(void)
{
    int i;
    int height[] = {178, 175, 173, 165, 179};
    char name[][20] = {"Sato", "Sanaka", "Takao", "Mike", "Masaki"};

    sort(height, name, NUMBER);

    for (i = 0; i < NUMBER; i++)
        printf("%2d: %-8s%4d\n", i + 1, name[i], height[i]);

    return (0);
}

```

运行结果

```

1:Mike    165
2:Takao   173
3:Sanaka  175
4:Sato    178
5:Masaki  179

```

结构体

在刚才的程序中，分别定义了表示身高的数组 *height* 和表示名字的数组 *name*。由于这两个数组是相互独立的，所以很可能看不出 *height*[3] 与 *name*[3] 之间的关联性，从而也就不知道它们表示的是“同一名学生”的身高和名字了。

请想象在现实世界中我们是如何处理名字和身高的。假设现在要汇总学校的体检信息。我们会为名字、身高、体重等分别建表吗？显然不会。通常是每人发一张“体检卡”，在卡片上面记录名字、身高等信息（图 12-1）。如果一个班有 50 名学生，那么 50 张“体检卡”即为一个集合。

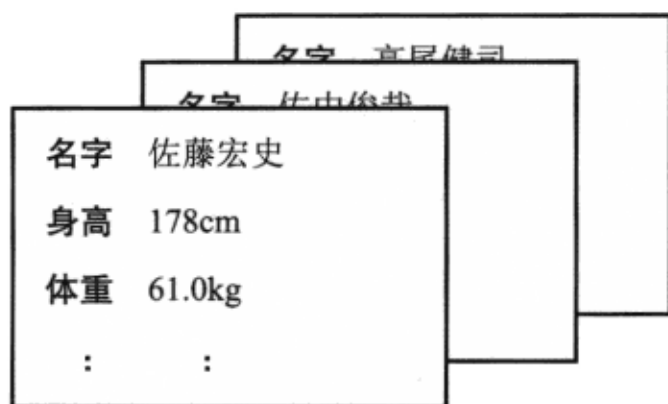


图 12-1 体检卡集合

在 C 语言中，像这种卡片形式的数据结构是通过**结构体**（structure）来实现的。

■ 注意 ■

结构体是聚合了一系列数据的数据结构。

可以像右图那样进行结构体的声明，该结构体中包含了字符数组（字符串）型的名字、**int** 型的身高、**float** 型的体重以及 **long** 型的奖学金。

其中，结构体的名字 *gstudent* 称为**结构名**（structure tag）。{ } 中声明的 *name*、*height* 等称为**结构体成员**（member）。

```
struct gstudent {  
    char    name[20];    /* 名字 */  
    int     height;      /* 身高 */  
    float   weight;      /* 体重 */  
    long    schols;      /* 奖学金 */  
};
```

图 12-2 结构体的声明

► 在第 8 章讲到**枚举型**的枚举名时也遇到过“tag”一词。与枚举型一样，在结构体声明的末尾也要加上分号；。

这个声明用来告诉程序“**struct gstudent** 是这样的类型哦”，并不定义对象（变量）的实体。即如图 12-3 所示，只是描画了卡片格式的框架。

要保存名字、身高等数据，还需像下述语句那样声明和定义 **struct gstudent** 型的实体对象（变量）。

```
struct gstudent shibata;
```

结构体的声明及该类型对象的定义，通常如图 12-4 所示。

图 12-3 结构体的框架

名 字	
身 高	
体 重	
奖学金	

一旦声明了结构体，就能在程序中自由地使用“**struct 结构名**”的结构体类型。

```
struct test {  
    int    x;  
    long   y;  
    double z;  
} ta, tb;
```

► 在声明结构体类型时，也可以同时定义该类型的对象。以图 12-4 为例，可以像右图那样进行声明和定义。

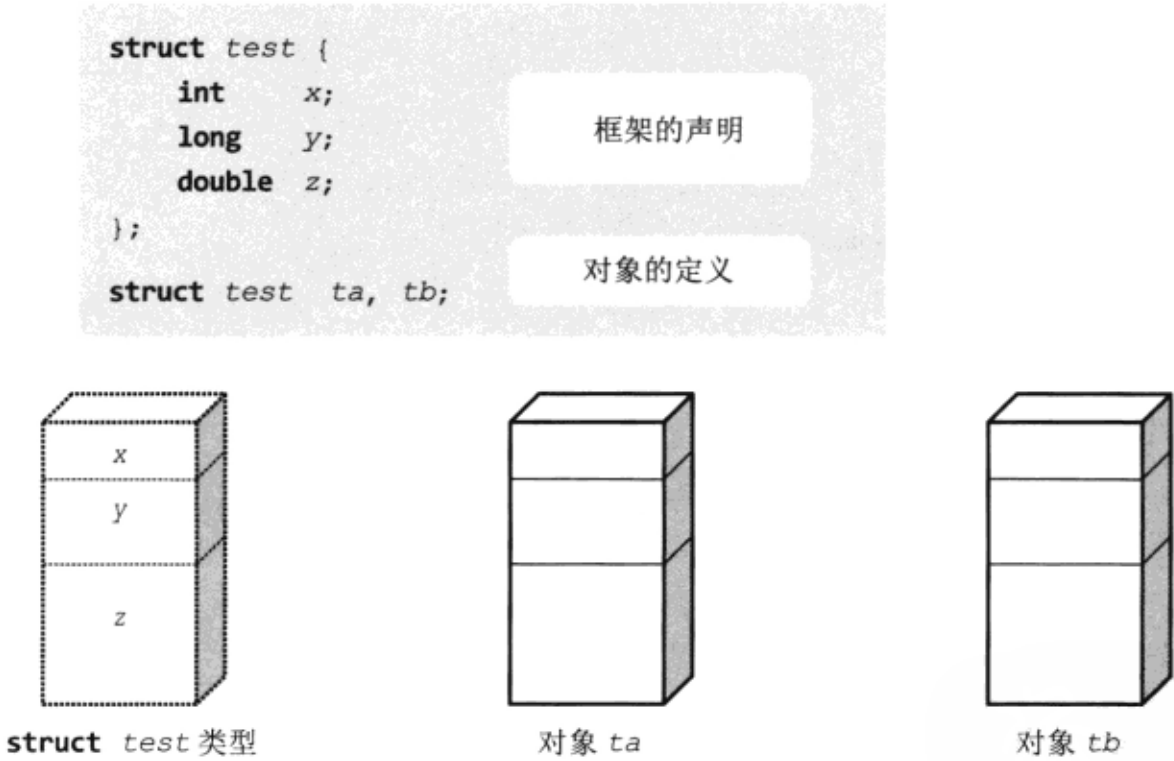


图 12-4 结构体的声明和对象的定义

也可以在声明中省略结构名。例如

```
struct {  
    /* 中略 */  
} a, b;
```

其中将 *a* 和 *b* 定义为此处声明的结构体类型的对象。但由于该结构体是匿名的，所以程序的其他地方就不能方便地声明相同类型的结构体。

结构体成员（. 运算符）

我们用 **struct** *gstudent* 结构体类型来表示佐中的名字、身高、奖学金数据。程序如代码清单 12-3 所示。

代码清单 12-3

```
/*
    用表示学生的结构体来显示佐中的信息
*/

#include <stdio.h>
#include <string.h>

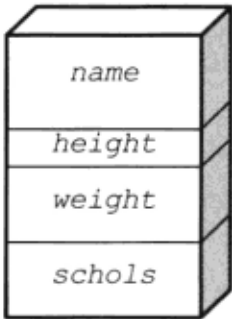
struct gstudent {
    char    name[20];          /* 姓名 */
    int     height;            /* 身高 */
    float   weight;            /* 体重 */
    long    schols;            /* 奖学金 */
};

int main(void)
{
    struct gstudent sanaka;

    strcpy(sanaka.name, "Sanaka"); /* 姓名 */
    sanaka.height = 175;           /* 身高 */
    sanaka.weight = 60.5;          /* 体重 */
    sanaka.schols = 70000;         /* 奖学金 */

    printf("姓 名 = %s\n", sanaka.name);
    printf("身 高 = %d\n", sanaka.height);
    printf("体 重 = %f\n", sanaka.weight);
    printf("奖学金 = %ld\n", sanaka.schols);

    return (0);
}
```



运行结果

姓 名 = Sanaka
身 高 = 175
体 重 = 60.500000
奖学金 = 70000

在下述表达式中

sanaka.height 对象名.成员名

可以用被称为句点运算符的 **. 运算符**（. operator）来表示结构体对象的成员（表 12-1）。因此，可以像图 12-5 那样访问对象 *sanaka* 的各个成员。
sanaka.height 是 **int** 型对象，所以它和普通的 **int** 型变量一样可以进行赋值和取值操作。

■ 表 12-1 . 运算符

. 运算符	<i>a . b</i>	表示结构体 <i>a</i> 的成员 <i>b</i> 。
-------	--------------	-------------------------------

成员的初始化

只要是用到的对象，除特殊要求外一般应作初始化。如代码清单 12-4 所示为改写后的程序，程序中对佐中的数据进行了初始化。

代码清单 12-4

```

/*
    对表示学生的结构体的成员佐中进行初始化
*/

#include <stdio.h>

struct gstudent {
    char  name[20];    /* 姓名 */
    int   height;      /* 身高 */
    float weight;      /* 体重 */
    long  schols;      /* 奖学金 */
};

int main(void)
{
    struct gstudent  sanaka  = {"Sanaka", 175, 60.5};

    printf("姓 名 = %s\n",  sanaka.name);
    printf("身 高 = %d\n",  sanaka.height);
    printf("体 重 = %f\n",  sanaka.weight);
    printf("奖学金 = %ld\n", sanaka.schols);

    return (0);
}
    
```

运行结果	
姓 名 =	Sanaka
身 高 =	175
体 重 =	60.500000
奖学金 =	0

为结构体赋初始值的形式与数组相同。各个结构体成员的初始值依次排列在 { } 里面，并用逗号进行分割。

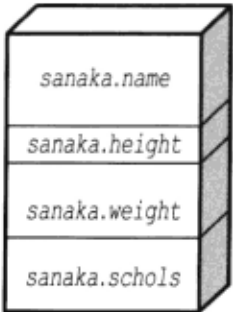


图 12-5 结构体对象的成员

再则，未赋初始值的元素被初始化为 0，这一点也和数组相同。在该程序中，未定义奖学金 `sanaka.schols` 的初始值，所以奖学金的值为 0，这从运行结果中也可以看出。

► 如图 12-5 所示，在结构体声明中写在前面的成员的地址较低（小）。也就是说编译器是根据声明的顺序在内存单元中保存结构体成员的。为各个结构体成员定义初始值的顺序也是如此。

结构体成员（-> 运算符）

还记得第10章指针示例程序中的 *hiroko* 函数吗？那时提到洋子的恋爱要求比较高，她还拥有超能力，如果恋人的身高低于 180cm，就能将他变为 180cm。

现在我们就使用 *gstudent* 结构体来改写那个 *hiroko* 函数。对了，洋子也不喜欢太胖的人，她还有一个超能力，那就是如果体重超过 80kg 就能将他变为 80kg。程序见代码清单 12-5。

hiroko 函数的形参 *std* 负责接收指向结构体 *gstudent* 的指针。

该函数中分别通过以下形式表示身高和体重。

```
(*std).height    /* 身高 */
(*std).weight    /* 体重 */
```

我们结合图 12-6 来理解一下吧。

hiroko 函数的形参 *std* 接收的是指向保存佐中数据的结构体 *sanaka* 的指针（图中的地址为 300 号）。

在指针变量前加上指针运算符 ***，就表示该指针指向的对象实体，即 **std* 是 *sanaka* 的别名。所以，通过 **std* 可以表示其指向的对象的身高成员 *(*std).height* 和体重成员 *(*std).weight*。

► 不能用 **std.height* 来表示 **std* 的身高成员。因为 *.* 运算符的优先级比指针运算符 *** 高，表达式会被解释成 **(std.height)*，产生语法错误。

当然，*(*std).height* 和 *(*std).weight* 的写法比较麻烦，很容易漏写前面的括号。不过素以简洁著称的 C 语言可不会有此疏漏。使用 **-> 运算符** (*-> operator*) 就可以将 *(*std).height* 简写为

```
std->height    指向结构体的指针 -> 成员名
```

■ 注意 ■

对于指针 *ptr* 指向的结构体的成员 *mem*，推荐使用 *->* 运算符将 *(*ptr).mem* 简写为 *ptr->mem*。

■ 表 12-2 -> 运算符

-> 运算符	<i>a -> b</i>	用指针访问结构体 <i>a</i> 中的成员 <i>b</i> 。
------------------	------------------	-----------------------------------

► *.* 运算符形如句点，所以称为**句点运算符**。而 *->* 运算符形如箭头，则称为**箭头运算符**。

代码清单 12-5

```

/*
    拥有超能力的洋子
*/

#include <stdio.h>

struct gstudent {
    char    name[20];    /* 姓名 */
    int     height;      /* 身高 */
    float   weight;      /* 体重 */
    long    schols;      /* 奖学金 */
};

/* --- 能改变人身高和体重的洋子 --- */
void hiroko(struct gstudent *std)
{
    if ((*std).height < 180) (*std).height = 180;
    if ((*std).weight > 80) (*std).weight = 80;
}

int main(void)
{
    struct gstudent sanaka = {"Sanaka", 175, 60.5, 70000};

    hiroko(&sanaka);

    printf("姓 名 = %s\n",    sanaka.name);
    printf("身 高 = %d\n",    sanaka.height);
    printf("体 重 = %f\n",    sanaka.weight);
    printf("奖学金 = %ld\n",  sanaka.schols);

    return (0);
}

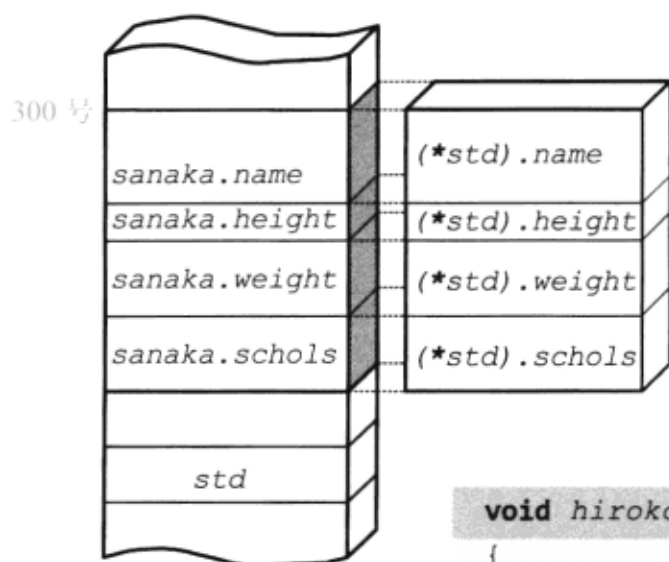
```

运行结果

```

姓 名 = Sanaka
身 高 = 180
体 重 = 60.500000
奖学金 = 70000

```



```

int main(void)
{
    struct gstudent sanaka;
    :
    hiroko(&sanaka)
    :
}

void hiroko(struct gstudent * std)
{
    if ((*std).height < 180) (*std).height = 180;
    if ((*std).weight > 80) (*std).weight = 80;
}

```

300 号

图 12-6 指针指向的结构体的成员

结构体和 typedef

第7章学习过 **typedef** 声明，它可以给原有的数据类型定义“同义词”，它的作用等同于数据类型名称。有效利用 **typedef** 声明，可以简化下述冗长的写法。

struct *gstudent*

改写后的程序见代码清单 12-6。

代码清单 12-6

```

/*
    拥有超能力的洋子（其二）
*/
#include <stdio.h>

/*--- 表示学生的结构体 ---*/
typedef struct {
    char    name[20];    /* 姓名 */
    int     height;      /* 身高 */
    float   weight;      /* 体重 */
    long    schols;      /* 奖学金 */
} student;

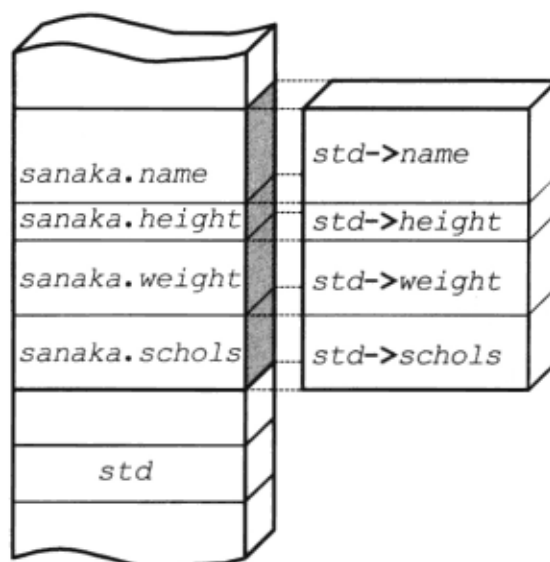
/*--- 能够改变人身高和体重的洋子 ---*/
void hiroko(student *std)
{
    if (std->height < 180) std->height = 180;
    if (std->weight > 80) std->weight = 80;
}

int main(void)
{
    student sanaka = {"Sanaka", 175, 60.5, 70000};

    hiroko(&sanaka);

    printf("姓 名 = %s\n",    sanaka.name);
    printf("身 高 = %d\n",    sanaka.height);
    printf("体 重 = %f\n",    sanaka.weight);
    printf("奖学金 = %ld\n",  sanaka.schols);

    return (0);
}
    
```



运行结果

```

姓 名 = Sanaka
身 高 = 180
体 重 = 60.500000
奖学金 = 70000
    
```

在前几页的结构体中，结构名为 *gstudent*，数据类型名称为“**struct** *gstudent*”。而本例为结构体定义了一个名字 *student*，“*student*”即为数据类型名称。

► 本例与上一页的程序不同，使用的不是句点运算符而是箭头运算符。

结构体和程序

在表示人的身高时，通常使用 **int** 型或 **double** 型对象。这时，为了在程序世界中表示人的身高这一现实世界对象（物），需要将它对应至对象（变量）并为它定义一个名字 *height*（图 12-7）。



图 12-7 整型对象

如果程序中还需要“体重”，那么也同样需要进行对应，例如定义一个名为 *weight* 的对象。

毋庸置疑，现实世界和程序世界显然是不同的。因此，我们所关注的“身高”、“体重”等属性，在程序中是用 *height*、*weight* 等变量来实现的。

本章学习的结构体，不仅关注人的某一个属性，而且关注其他多个属性；即并非分别处理身高、体重等数据，而是将“身高的对象”、“体重的对象”等对象聚合为一个对象进行表示（图 12-8）。

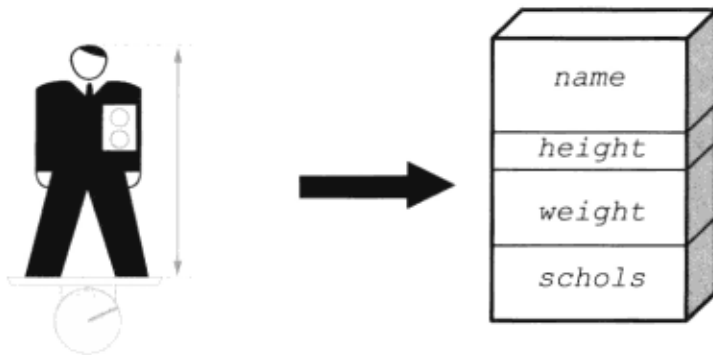


图 12-8 结构体对象

将现实世界与程序世界对应起来的时候，对应方法会因问题的类型和范围而各不相同，但是比较自然的做法是遵循“聚合应聚合的对象”这一方针。

使用结构体可以使程序变得简洁明了。

聚合类型

数组和结构体在处理多个对象的集合方面具有诸多相同点，它们统称为**聚合类型**（aggregate type）。

下面，我们来看看数组与结构体有哪些不同点。

■ 元素类型

数组用于高效地操作“相同类型”数据的集合。而结构体这种数据结构通常用于高效地操作“不同类型”数据的集合（当然，偶尔也会有成员类型全部相同的情况）。

■ 可否赋值

即便两个数组的元素数相同，也不能相互赋值。

但是，相同类型的结构体可以相互赋值。如右图所示，*gy* 中的所有成员都赋给了 *gx* 中相应的成员。

```
int  a[6], b[6];
a = b;           /* 错误 */

struct gstudent  gx, gy;
gx = gy;         /* OK */
```

命名空间

第 8 章曾简单地介绍了**命名空间**（name space）。只要命名空间不同，就可以使用拼写相同的标识符（名字）。命名空间的分类有以下四种：

（1）标签（label）名

（2）小标签（tag）名

（3）成员名

（4）一般性标识符

※ 函数名、对象名等

在右图所示的程序中，*x* 分别用作小标签名、成员名、对象（变量）名、标签名。像这样，只要不属于同一个命名空间，即使在同一有效范围内使用相同的名字，也不会产生任何问题。

```
#include <stdio.h>

int main(void)
{
    struct x {                /* 小标签名 */
        int x;                /* 成员名 */
        int y;                /* 变量名 */
    } x, y;                    /* 变量名，成员名 */
    x.x = 5;                   /* 标签名 */
x:                             /* 变量名，成员名 */
    printf("%d\n", x.x);

    return (0);
}
```

返回结构体的函数

可以相互赋值的结构体，也可以用作函数的返回值类型（数组则不行）。请看代码清单 12-7 的程序。

`set_xyz` 函数将形参 `x`、`y`、`z` 接收到的值赋给结构体 `temp`，再将该结构体的值原样返回。请注意程序中蓝色底纹部分。`set_xyz` 函数返回的结构体被直接赋给了变量 `xyz`。

代码清单 12-7

```
/*
    返回结构体的函数
*/
#include <stdio.h>

struct xyz {
    int    mx;
    long   my;
    double mz;
};

/* --- 返回结构体 xyz --- */
struct xyz set_xyz(int x, long y, double z)
{
    struct xyz temp;

    temp.mx = x;
    temp.my = y;
    temp.mz = z;
    return (temp);
}

int main(void)
{
    struct xyz xyz = {0, 0, 0};

    xyz = set_xyz(10, 320, 35.6);

    printf("xyz.mx = %d\n",      xyz.mx);
    printf("xyz.my = %ld\n",     xyz.my);
    printf("xyz.mz = %f\n",      xyz.mz);

    return (0);
}
```

运行结果

```
xyz.mx = 10
xyz.my = 320
xyz.mz = 35.600000
```

结构体数组

先回到本章开头的问题，即用聚合了姓名、身高、体重、奖学金数据的结构体表示 5 名学生的信息，并按身高进行升序排列。

要表示 5 名学生的信息，可以像图 12-9 那样定义元素类型为结构体的数组。程序如代码清单 12-8 所示。

注意看 *swap* 函数。现在已经不需要为交换身高和名字而分别编写函数了，只通过一个函数就能实现该功能。

该函数可以直接交换指针 *x* 和指针 *y* 指向的 *student* 类型的结构体内容。

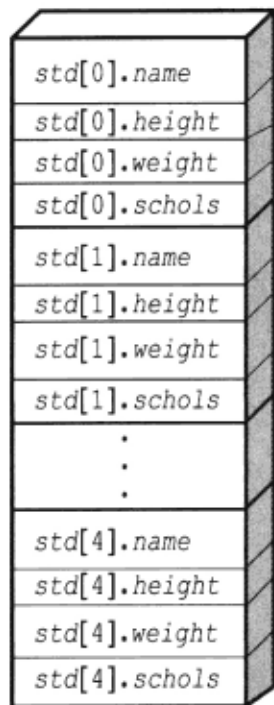


图 12-9 结构体数组

派生类型

“结构体”聚合了各种类型的对象。这里创建了结构体集合的“数组”。像这样，在 C 语言中可以组合各种方法创建出无穷的数据类型。

通过这种方式创建的数据类型称为**派生类型**（derived type）。能够通过派生创建的类型如下（可以自由组合）：

数组类型（array type）

将某一种元素类型对象的集合分配在连续的内存单元中（第 5 章）。

结构体类型（structure type）

按成员的声明顺序分配内存单元。各成员的数据类型可以不同（本章）。

共用体类型（union type）

不同的成员可以放入同一段内存单元，使之相互重叠。

函数类型（function type）

由 1 个返回类型和 0 个以上的形参及其数据类型构成（第 6 章）。

指针类型（pointer type）

创建为指向对象或函数的数据类型（第 10 章）。

► 有关共用体和指向函数的指针的知识，超过了本书的范围，就不详细展开了。

代码清单 12-8

```

/*
    将 5 名学生按身高进行升序排列
*/

#include <stdio.h>

#define NUMBER      5  /* 学生人数 */

typedef struct {
    /* 省略 (参考代码清单 12-6) */
} student;

/*--- 对 x 和 y 指向的学生进行交换 ---*/
void swap(student *x, student *y)
{
    student temp = *x;
    *x = *y;
    *y = temp;
}

/*--- 对数组 data 的前 n 个元素按身高进行升序排列 ---*/
void sort(student data[], int n)
{
    int k = n - 1;
    while (k >= 0) {
        int i, j;
        for (i = 1, j = -1; i <= k; i++)
            if (data[i - 1].height > data[i].height) {
                j = i - 1;
                swap(&data[i], &data[j]);
            }
        k = j;
    }
}

int main(void)
{
    int i;
    student std[] = {
        { "Sato", 178, 61.0, 80000}, /* 佐藤 */
        { "Sanaka", 175, 60.5, 70000}, /* 佐中 */
        { "Takao", 173, 80.0, 0}, /* 高尾 */
        { "Mike", 165, 72.0, 70000}, /* 平木 */
        { "Masaki", 179, 77.5, 70000}, /* 真崎 */
    };

    sort(std, NUMBER);

    puts("-----");
    for (i = 0; i < NUMBER; i++)
        printf("%-8s %6d%6.1f%7ld\n",
            std[i].name, std[i].height, std[i].weight, std[i].schols);
    puts("-----");

    return (0);
}

```

运行结果

```

-----
Mike      165  72.0  70000
Takao     173  80.0      0
Sanaka    175  60.5  70000
Sato      178  61.0  80000
Masaki    179  77.5  70000
-----

```

表示日期和时间的结构体

代码清单 12-9 所示为显示当天日期的程序。

`time` 函数用于获取当前日期和时间。

time	
头文件	<code>#include <time.h></code>
原 型	<code>time_t time(time_t *timer)</code>
说 明	求出日期时间。
返回值	返回当前的日期时间。若日期时间无效，则返回 -1。若 <code>timer</code> 不为 NULL，则在 <code>timer</code> 指向的对象中也保存日期时间。

参数（和返回值）所用的 `time_t` 类型，定义在 `<time.h>` 头文件中。使用不同的编译器该定义会略有不同，大致定义为：

```
typedef unsigned long time_t;
```

通过调用 `time` 函数，`time_t` 类型的变量 `current` 中就保存了当前时间。

```
time (&current);
```

► 在许多编译器中，保存的是从基准时间（例如 1970 年 1 月 1 日上午 0 时 0 分 0 秒）经过的秒数。

`localtime` 函数可以将 `time_t` 类型的时间转换为年、月、日、时、分、秒等我们日常生活中使用的时间形式。

localtime	
头文件	<code>#include <time.h></code>
原 型	<code>struct tm *localtime(const time_t *timer)</code>
说 明	通过时间结构体类型将日期时间转换为相应的本地时间。
返回值	返回指向转换后时间的指针。

struct tm	
头文件	<code>#include <time.h></code>
原 型	<pre>struct tm { int tm_sec; /* 秒 -[0, 61] */ int tm_min; /* 分 -[0, 59] */ int tm_hour; /* 时 -[0, 23] */ int tm_mday; /* 日 -[1, 31] */ int tm_mon; /* 距离 1 月份的月数 -[0, 11] */ int tm_year; /* 距离 1900 年的年数 */ int tm_wday; /* 距离星期日的天数 -[0, 6] */ int tm_yday; /* 距离 1 月 1 日的天数 -[0, 365] */ int tm_isdst; /* 夏令时旗标 */ }</pre>
说 明	保存日期时间的时间结构体类型。

代码清单 12-9

```

/*
    显示当天日期
*/

#include <time.h>
#include <stdio.h>

void put_date(void)
{
    time_t    current;
    struct tm  *local;
    char  wday_name[][7] = { "日", "一", "二", "三", "四", "五", "六" };

    time(&current);          /* 取得当前时间 */
    local = localtime(&current); /* 转换为本地时间的结构体 */
    printf("%4d 年 %02d 月 %02d 日 (%s)", local->tm_year + 1900, /* 年 */
        , local->tm_mon + 1, /* 月 */
        , local->tm_mday, /* 日 */
        , wday_name[local->tm_wday] /* 星期 */
    );
}

int main(void)
{
    printf("今天是 ");
    put_date();
    printf(".\n");

    return (0);
}

```

运行结果

今天是 1998 年 11 月 18 日 (星期三)。

► 在绝大多数编译器和运行环境中，会显示运行程序时的日期。

localtime 函数会返回转换后的 **struct tm** 类型对象的地址。该地址保存在指针 *local* 中。因此，*local->tm_year* 就是今年的公历年份减去 1900 之后的值，*local->tm_mday* 就是该日。

► 转换后的 **struct tm** 类型对象已由 **localtime** 函数定义（在编写的程序中不能自行定义）。

● 练习 12-1

编写如下函数，显示当前时间。

```
void put_time(void) { /* ... */ }
```

● 练习 12-2

发现身边可以用结构体表示的事物，并用程序实现。

12-2 作为成员的结构体

表示坐标的结构体

代码清单 12-10 所示的程序定义了由 X 坐标和 Y 坐标定位的点的结构体，并使用该结构体计算两点之间的距离。

代码清单 12-10

```
/*
 * 计算两点之间的距离
 */

#include <math.h>
#include <stdio.h>

#define sqr(n)      ((n) * (n))      /* 计算乘方 */

typedef struct {          /*=== 点 ===*/
    int x;                /* X坐标 */
    int y;                /* Y坐标 */
} point;

/* --- 返回点 pa 和点 pb 之间的距离 --- */
double distanceof(point pa, point pb)
{
    return (sqrt(sqr(pa.x - pb.x) + sqr(pa.y - pb.y)));
}

int main(void)
{
    point p1, p2;

    printf("点 1 的 X 坐标: ");    scanf("%d", &p1.x);
    printf("      Y 坐标: ");      scanf("%d", &p1.y);
    printf("点 2 的 X 坐标: ");    scanf("%d", &p2.x);
    printf("      Y 坐标: ");      scanf("%d", &p2.y);

    printf("两点之间的距离为 %.2f。 \n", distanceof(p1, p2));

    return (0);
}
```

运行结果

点 1 的 X 坐标: 10
 Y 坐标: 20
 点 2 的 X 坐标: 15
 Y 坐标: 35
 两点之间的距离为 15.81。

表示具有定位功能的汽车的结构体

下一页代码清单 12-11 所示的程序，可以计算汽车燃料和行驶距离。结构体 *car* 表示汽车，结构体 *point* 表示坐标，*point* 为 *car* 的一个成员。像这样，结构体的成员不限于基本的 **int** 型或 **double** 型，也可以是结构体类型、数组类型或枚举型。

代码清单 12-11

```

/*
  汽车行驶与燃料消耗
*/
#include <math.h>
#include <stdio.h>

#define sqr(n)      ((n) * (n))

typedef struct {
    /* 省略 (参考代码清单 12-10) */
} point;

typedef struct {          /*=== 汽车 ===*/
    double fuel;          /* 剩余燃料 */
    point pt;              /* 当前位置 */
} car;

/* --- 显示当前位置和剩余燃料 --- */
void put_info(car c)
{
    printf("当前位置: (%d,%d)\n", c.pt.x, c.pt.y);
    printf("剩余燃料: %.2f 升\n", c.fuel);
}

/* --- 向 X 和 Y 方向行驶 (dx,dy) 距离 --- */
int move(car *c, int dx, int dy)
{
    double dist = sqrt(sqr(dx) + sqr(dy));          /* 行驶距离 */
    if (dist > c->fuel)
        return (0);                                /* 无法行驶 */
    c->pt.x += dx;    c->pt.y += dy;    c->fuel -= dist;
    return (1);                                         /* 成功行驶 */
}

int main(void)
{
    car mycar = {90.0, {0, 0}};

    while (1) {
        int slct;
        int dx, dy;    /* 行驶距离 */

        put_info(mycar);
        printf("开动汽车吗 [ Yes...1 / No...0 ]: ");
        scanf("%d", &slct);
        if (slct != 1) break;
        printf("X 方向的行驶距离: ");    scanf("%d", &dx);
        printf("Y 方向的行驶距离: ");    scanf("%d", &dy);

        if (!move(&mycar, dx, dy))
            puts("\a 燃料不足无法行驶。");
    }
    return (0);
}

```

运行结果

```

当前位置: (0,0)
剩余燃料: 90.00 升
开动汽车吗【Yes...1 / No...0】: 1
X 方向的行驶距离: 10
Y 方向的行驶距离: 10
当前位置: (10,10)
剩余燃料: 75.86 升
开动汽车吗【Yes...1 / No...0】: 1
X 方向的行驶距离: 30
Y 方向的行驶距离: 40
当前位置: (70,50)
剩余燃料: 3.75 升
开动汽车吗【Yes...1 / No...0】: 1
X 方向的行驶距离: 55
Y 方向的行驶距离: 30
燃料不足无法行驶。
当前位置: (70,50)
剩余燃料: 3.75 升
开动汽车吗【Yes...1 / No...0】: 0

```

1

2

第 13 章

文件处理

好不容易在程序中完成了计算和字符串处理等操作，但随着程序运行结束，运行结果也跟着消失了，这样岂不可惜？

本章就来学习与数据持久化保存相关的文件处理基础知识。



13-1 文件与流

文件

程序的处理结果或计算结果会随着程序运行结束而消失。因此要将程序运行结束后仍需保存的数值和字符串等数据保存在文件（file）中。

流

针对文件、键盘、显示器、打印机等外部设备的数据读写操作都是通过流（stream）进行的。我们可以将流想象成流淌着字符的河。

由此可见，在前几章的学习中所有用到 **printf** 函数或 **scanf** 函数的程序都使用了流。

图 13-1 是流和输入输出的示意图。

printf 函数将字符 'A'、'B'、'C' 输出到连接显示器的流。

而从键盘输入的字符会进入流中，**scanf** 函数会将它们取出来，并将它们的值保存至变量 *x*。

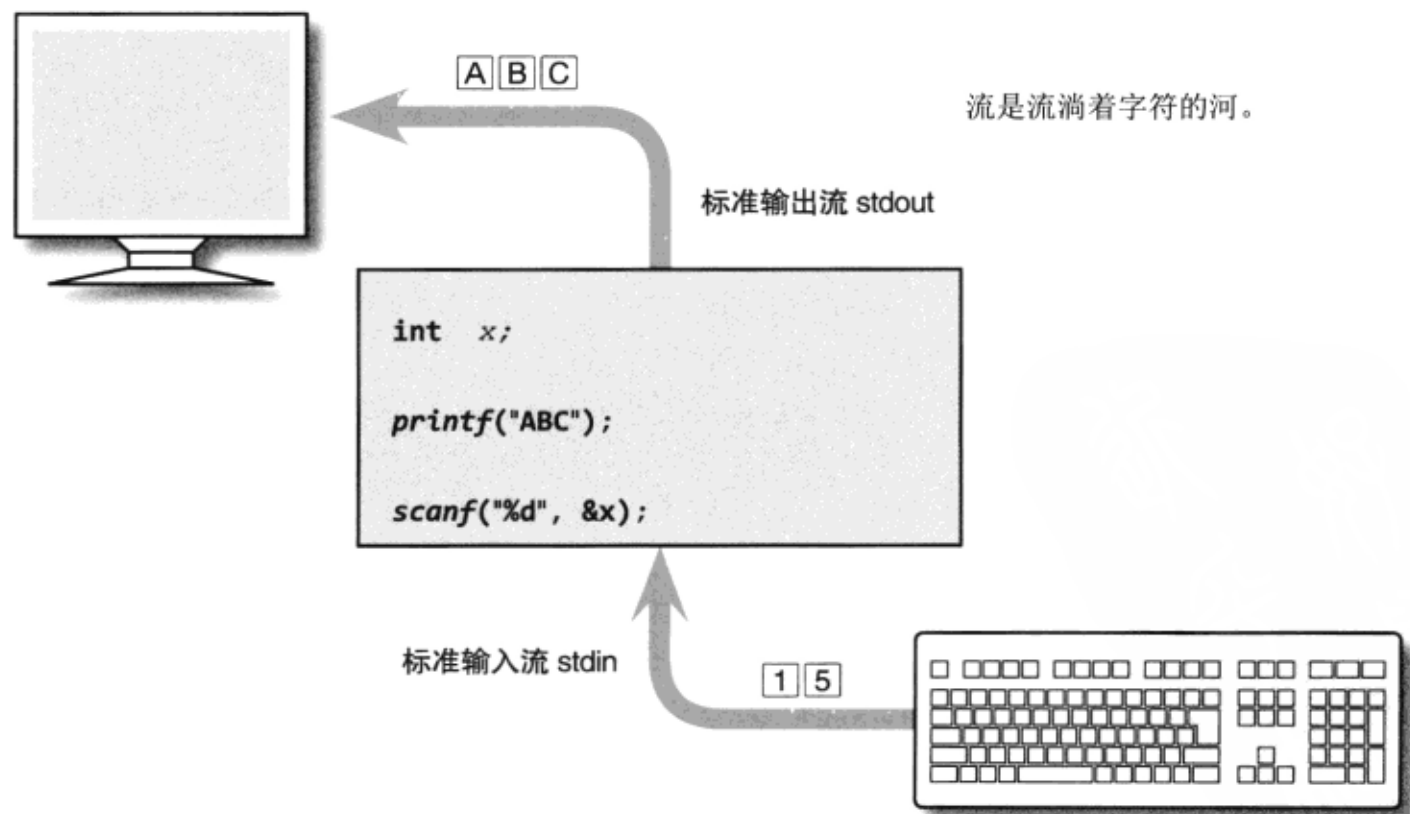


图 13-1 流和输入输出

标准流

我们之所以能够如此简单方便地执行使用了流的输入输出操作，是因为 C 语言程序在启动时已经将**标准流**（standard stream）准备好了。

标准流有以下三种。

■ **stdin** —— 标准输入流（standard input stream）

用于读取普通输入的流。在大多数环境中为从键盘输入。**scanf** 与 **getchar** 等函数会从这个流中读取字符。

■ **stdout** —— 标准输出流（standard output stream）

用于写入普通输出的流。在大多数环境中为输出至显示器界面。**printf**、**puts** 与 **putchar** 等函数会向这个流写入字符。

■ **stderr** —— 标准错误流（standard error stream）

用于写出错误的流。在大多数环境中为输出至显示器界面。

FILE 型

表示标准流的 **stdin**、**stdout**、**stderr** 都是指向 **FILE** 型的指针型。**FILE** 型是在 `<stdio.h>` 头文件中定义的，该数据类型用于记录控制流所需要的信息，其中包含以下数据：

■ 文件位置指示符（file position indicator）

记录当前访问地址。

■ 错误指示符（error indicator）

记录是否发生了读取错误或写入错误。

■ 文件结束指示符（end-of-file indicator）

记录是否已到达文件末尾。

通过流进行的输入输出都是根据上述信息执行操作的。而且这些信息也会随着操作结果更新。**FILE** 型的具体实现方法因编译器而异，一般多以结构体的形式实现。

打开文件

大家在使用纸质笔记本时通常都是先打开，然后再翻页阅读或在适当的地方书写。

程序中的文件处理过程也同样如此。首先打开文件并定位到文件开头，然后找到要读取或写入的目标位置进行读写操作，最后将文件关闭。

打开文件的操作称为**打开**（open）。函数库中的 **fopen** 函数用于打开文件，请看下一页中的函数说明。

该函数需要两个参数（图 13-2）。第 1 个参数是要打开的文件名，第 2 个参数是文件类型及打开模式。

► 文件类型有两种，即文本文件和二进制文件。本节先学习文本文件，下一节再学习二进制文件。

fopen 函数会为要打开的文件新建一个流，然后返回一个指向 **FILE** 型对象的指针，该 **FILE** 型对象中保存了控制这个流所需要的信息。

文件一旦打开后，就可以通过 **FILE *** 型指针对流进行操作。

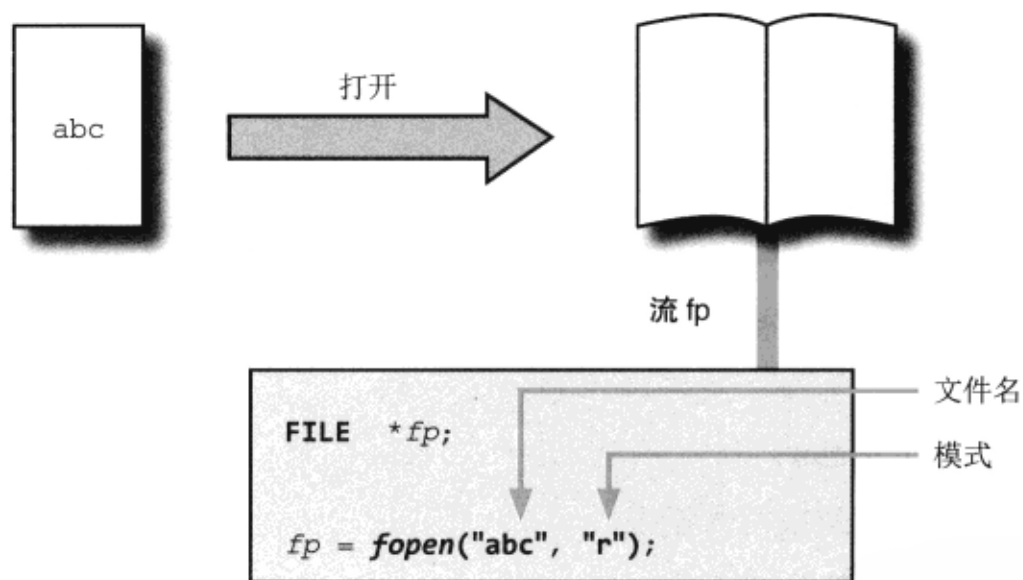


图 13-2 文件的打开

和程序启动时便准备好的标准流不同，要打开文件时必须先在程序中定义 **FILE *** 型的指针变量。然后将 **fopen** 函数返回的指针赋于该变量，就可以通过该指针变量对文件进行操作了。

变量可以任意命名，这里我们将其命名为 *fp*。*fp* 不是流的实体，而是指向流的指针，严格来讲应称之为“指针 *fp* 所指向的流”，本书为简单起见称为“流 *fp*”。

fopen	
头文件	<code>#include <stdio.h></code>
原型	<code>FILE *fopen(const char *filename, const char *mode);</code>
说明	<p>打开文件名为<code>filename</code>所指字符串的文件，并将该文件与流相关联。实参<code>mode</code>指向的字符串，以下述字符序列中的某一项开头。</p> <p><code>r</code> 以只读模式打开文本文件。</p> <p><code>w</code> 以只写模式建立文本文件，若文件存在则文件长度清为0。</p> <p><code>a</code> 以追加模式（从文件末尾处开始的只写模式）打开或建立文本文件。</p> <p><code>rb</code> 以只读模式打开二进制文件。</p> <p><code>wb</code> 以只写模式建立二进制文件，若文件存在则文件长度清为0。</p> <p><code>ab</code> 以追加模式（从文件末尾处开始的只写模式）打开或建立二进制文件。</p> <p><code>r+</code> 以更新（读写）模式打开文本文件。</p> <p><code>w+</code> 以更新模式建立文本文件，若文件存在则文件长度清为0。</p> <p><code>a+</code> 以追加模式（从文件末尾处开始写入的更新模式）打开或建立文本文件。</p> <p><code>r+b</code>或<code>rb+</code> 以更新（读写）模式打开二进制文件。</p> <p><code>w+b</code>或<code>wb+</code> 以更新模式建立二进制文件，若文件存在则文件长度清为0。</p> <p><code>a+b</code>或<code>ab+</code> 以追加模式（从文件末尾处开始写入的更新模式）打开或建立二进制文件。</p>
	<p>以读取模式（<code>mode</code>以字符'<code>r</code>'开头）打开文件时，如果该文件不存在或者没有读取权限，则文件打开失败。</p> <p>对于以追加模式（<code>mode</code>以字符'<code>a</code>'开头）打开的文件，打开后的写入操作都是从文件末尾处开始的。此时<code>fseek</code>函数的调用会被忽略。在有些用<code>null</code>字符填充二进制文件的编译器中，以追加模式（<code>mode</code>以字符'<code>a</code>'开头，并且第2或第3个字符是'<code>b</code>'）打开二进制文件时，会将流的文件位置指示符设为超过文件中数据末尾的位置。</p> <p>对于以更新模式（<code>mode</code>的第2或第3个字符为'<code>+</code>'）打开的文件相关联的流，可以进行输入和输出操作。但若要在输出操作之后进行输入操作，就必须在这两个操作之间调用文件定位函数（<code>fseek</code>、<code>fsetpos</code>或<code>rewind</code>）。除非输入操作检查到文件末尾，其他情况下若要在输入操作之后进行输出操作，也必须在这两个操作之间调用文件定位函数。有些编译器会将以更新模式打开（或建立）文本文件改为以相同模式打开（或建立）二进制文件，这不会影响操作。</p> <p>当能够识别到打开的流没有关联通信设备时，该流为全缓冲。打开时会清空流的错误指示符和文件结束指示符。</p>
返回值	返回一个指向对象的指针，该对象用于控制打开的流。打开操作失败时，返回空指针。

- 打开文件时可以指定以下四种模式：
- 只读模式 —— 只从文件输入。
 - 只写模式 —— 只向文件输出。
 - 更新模式 —— 既从文件输入，也向文件输出。
 - 追加模式 —— 从文件末尾处开始向文件输出。

关闭文件

当我们读完一本书时会将它合上，文件也同样如此。在文件使用结束后，就要断开文件与流的关联将流关闭。这个操作就称为关闭（close）文件。

以下是用于关闭文件的 **fclose** 函数说明^①。

fclose	
头文件	#include <stdio.h>
原 型	int fclose(FILE *stream);
说 明	刷新stream所指向的流，然后关闭与该流相关联的文件。流中留在缓冲区里面尚未写入的数据会被传递到宿主环境 ^① ，由宿主环境将这些数据写入文件。而缓冲区里面尚未读取的数据将被丢弃。然后断开流与文件的关联。如果存在系统自动分配的与该流相关联的缓冲区，则会释放该缓冲区。
返回值	若成功地关闭流，则返回0。检查到错误时返回EOF。

图 13-3 为关闭文件的示意图。只要将打开文件时 **fopen** 函数返回的指针传给 **fclose** 函数即可。

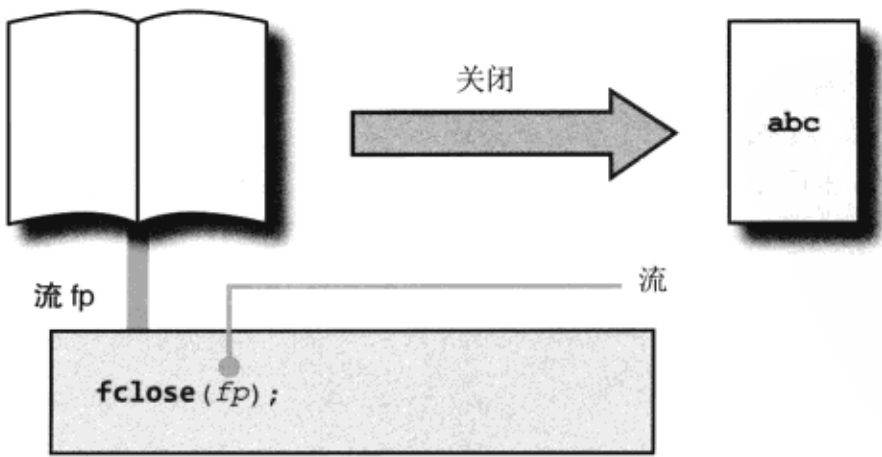


图 13-3 文件的关闭

① 即可使程序正常运行的计算机环境。

打开与关闭文件示例

代码清单 13-1 中的程序演示了如何通过调用 **fopen** 函数和 **fclose** 函数来打开和关闭文件。

代码清单 13-1

```
/*
  打开与关闭文件
*/

#include <stdio.h>

int main(void)
{
    FILE *fp;

    fp = fopen("abc", "r");           /* 打开文件 */

    if (fp == NULL)
        printf("\a文件打开失败.\n");
    else {
        /* 执行文件读入等操作 */
        fclose(fp);                  /* 关闭文件 */
    }

    return (0);
}
```

这个程序所做的工作是先以只读模式 **"r"** 打开名为 **"abc"** 的文件，然后将它关闭。当文件打开失败，**fopen** 函数返回 **NULL** 时，会显示“文件打开失败”。

► 该程序也可判断 **"abc"** 文件是否存在。

● 练习 13-1

代码清单 13-1 中的程序只能打开名为 **"abc"** 的文件。请将程序改为从键盘读入文件名，然后打开该文件。

● 练习 13-2

编写程序，从键盘读入文件名，若该文件存在则显示“该文件存在”，否则显示“该文件不存在”。

文件数据汇总

代码清单 13-2 所示的程序会将保存在文件中的姓名、身高、体重（个人信息）逐条读入并显示出来，最后还会显示平均身高和平均体重。

个人信息保存在图 13-4 所示的 "hw.dat" 文件中。

► 请在编辑器中输入图 13-4 的数据，并将文件命名为 "hw.dat" 保存至程序所在目录。

与上一页的程序相同，作为读写对象的文件必须和程序保存在相同的目录中。

Aiba 160 59.3
Kurata 162 51.6
Masaki 182 76.5
Tanaka 170 60.7
Tsuji 175 83.9
Washio 175 72.5

图 13-4 "hw.dat" 文件

变量 *ninzu* 保存人数（读入了几个人的数据），变量 *hsum* 和 *wsum* 分别保存身高合计和体重合计。这些变量都初始化为 0 或 0.0。

FILE * 型指针 *fp* 的声明以及打开与关闭文件的所有程序结构都和上一页的程序相同。

要从文件读取数据就需要使用 **fscanf** 函数了。**fscanf** 函数可以对任意流执行与 **scanf** 函数相同的输入操作。它比 **scanf** 函数多了 1 个参数，具体函数说明如下：

fscanf	
头文件	#include <stdio.h>
原 型	int fscanf(FILE *stream, const char *format, ...);
说 明	从stream指向的流（而不是从标准输入流）中读取数据。除此以外，与scanf函数完全相同。
返回值	若没有执行任何转换就发生了输入错误，则返回宏定义EOF的值。否则，返回成功赋值的输入项数。若在输入中发生匹配错误，则返回的输入项数会少于转换说明符对应的实参个数，甚至为0。

函数的用法很简单。例如，要从流 *fp* 中读取十进制的整数值并保存至变量 *x*，只需使用下述语句调用函数即可。

```
fscanf(fp, "%d", &x); /* 只比 scanf 函数多了 1 个参数！ */
```

与 **scanf** 函数相比，仅增加了第一个参数，即输入流。

★

本程序通过下述语句读取个人信息。

```
fscanf(fp, "%s%lf%lf", name, &height, &weight);
```

意思是从流 *fp* 中读取 1 个字符串和 2 个 **double** 型实数，分别将它们保存至 *name*、*height* 和 *weight*。

代码清单 13-2

```
/*
    读入身高和体重，计算并显示它们的平均值
*/
```

```
#include <stdio.h>
```

```
int main(void)
{
```

```
    FILE          *fp;
    int            ninzu = 0;           /* 人数 */
    char           name[100];          /* 姓名 */
    double         height, weight;     /* 身高，体重 */
    double         hsum = 0.0;         /* 身高合计 */
    double         wsum = 0.0;         /* 体重合计 */
```

```
    if ((fp = fopen("hw.dat", "r")) == NULL)      /* 打开文件 */
        printf("\a文件打开失败。\\n");
```

```
    else {
        while (fscanf(fp, "%s%lf%lf", name, &height, &weight) == 3) {
            printf("%-10s %5.1f %5.1f\\n", name, height, weight);
            ninzu++;
            hsum += height;
            wsum += weight;
        }
        printf("-----\\n");
        printf("平均      %5.1f %5.1f", hsum / ninzu, wsum / ninzu);
        fclose(fp);                          /* 关闭文件 */
    }
```

```
    return (0);
}
```

运行示例

```
Aiba      160.0  59.3
Kurata    162.0  51.6
Masaki    182.0  76.5
Tanaka    170.0  60.7
Tsuji     175.0  83.9
Washio    175.0  72.5
-----
平均      170.7  67.4
```

和 `scanf` 函数一样，`fscanf` 函数会返回读取到的项目数。当正常地读取到姓名、身高、体重项目返回 3 时，就会继续 `while` 语句循环直至读取不到个人信息（已读取完所有信息，或因出错而不能进行读取）。

在这个 `while` 语句中，首先会显示读取到的个人信息，然后让变量 `ninzu` 自增，最后将读取到的身高和体重累加到 `hsum` 和 `wsum`。

当读取不到三个项目时，`while` 语句就会结束循环，这时再显示身高和体重的平均值。

● 练习 13-3

改写代码清单 13-1 中的程序，将从文件读入的个人信息按身高排序后显示。

写入日期和时间

大家已经掌握了如何从文件读取，那么本节就来看看如何写入。

printf 函数是向标准输出流进行输出的函数，而向任意流执行同样操作的就是 **fprintf** 函数，它的说明如下。

fprintf	
头文件	<code>#include <stdio.h></code>
原型	<code>int fprintf(FILE *stream, const char *format, ...);</code>
说明	向 <i>stream</i> 指向的流（而不是标准输出流）写入数据。除此以外，与 printf 函数完全相同。
返回值	返回发送的字符数。当发生输出错误时，返回负值。

fprintf 函数的用法也很简单。例如，要向流 *fp* 写入整数 *x* 的十进制数值，只需使用下述语句调用函数即可。

```
fprintf(fp, "%d", x);          /* 只比 printf 函数多了 1 个参数! */
```

与 **printf** 函数相比，仅增加了第一个参数，即输出流。

*

在上一章的代码清单 12-9 中，我们学习了获取并显示当前日期的程序。本章就以此为参考，看看如何将程序运行时的日期和时间写入文件。请看代码清单 13-3 所示的程序。

FILE * 型指针 *fp* 的声明以及打开与关闭文件等程序结构都与之前的程序相同。唯一不同的一点是本程序是以只写模式 "**w**" 打开文件的。

► 由于是以只写模式打开文件，所以要注意此时如果存在同名文件，就会清空文件原来内容，只保存由本程序写入的内容。

下述代码负责将日期和时间写入文件。

```
fprintf(fp, "%d %d %d %d %d\n",  
        local->tm_year + 1990, local->tm_mon + 1, local->tm_mday,  
        local->tm_hour,      local->tm_min,      local->tm_sec );
```

公历年、月、日、时、分、秒是以十进制数写入的，所以程序运行以后 "**dt_dat**" 文件的内容如图 13-5 所示。



2004 7 10 13 21 5

图 13-5 "dt_dat" 文件

► 图中显示的数值仅供参考。实际写入文件的内容为程序运行时的日期和时间。

代码清单 13-3

```

/*
 向文件写出程序运行时的日期和时间
*/

#include <time.h>
#include <stdio.h>

int main(void)
{
    FILE *fp;
    time_t t;
    struct tm *local;

    time(&t);
    local = localtime(&t);

    if ((fp = fopen("dt_dat", "w")) == NULL)           /* 打开文件 */
        printf("\a文件打开失败.\n");
    else {
        printf("写出当前日期和时间.\n");
        fprintf(fp, "%d %d %d %d %d %d\n",
            local->tm_year + 1900, local->tm_mon + 1, local->tm_mday,
            local->tm_hour, local->tm_min, local->tm_sec );
        fclose(fp);                                     /* 关闭文件 */
    }

    return (0);
}

```

► 在第 1 节中提到过标准输入流 `stdin` 和标准输出流 `stdout` 都是指向 **FILE** 的指针型。因此，下面两条语句的功能相同，都是从标准输入流读取整数值，并保存至变量 `x`。

```

scanf("%d", &x);

fscanf(stdin, "%d", &x);           /* 等同于 scanf("%d", &x); */

```

同样，下面两条语句的功能也相同，它们都向标准输出流写入整数 `x` 的十进制数值。

```

printf("%d", x);

fprintf(stdout, "%d", x);          /* 等同于 printf("%d", x); */

```

● 练习 13-4

请采用代码清单 13-2 的文件写入形式，编写一个从键盘读取姓名、身高和体重，并将这些数据写入文件的程序。

获取上一次运行时的信息

我们将刚才的程序改得更实用一些，请看代码清单 13-4 中的程序。运行后会得到图 13-6 所示的结果。

(a) 程序第一次运行时的运行结果

运行结果

程序第一次运行。

(b) 程序自第二次起运行时的运行结果

运行结果

上一次运行是在2004年7月3日15时20分15秒。

图 13-6 代码清单 13-4 的运行结果

如果程序是第一次运行，就会显示第一次运行时所对应的消息。如果程序运行了两次以上，就会显示上一次运行时的日期和时间。

本程序中定义的 `get_data` 函数和 `put_data` 函数的功能如下：

■ `get_data` 函数

在程序开头调用。根据 "`datetime.dat`" 文件是否打开成功，执行下述分支处理。

● 打开失败时

判断为程序第一次运行，显示“程序第一次运行。”。

● 打开成功时

将程序上一次运行时写入的日期和时间读入文件并显示。

■ `put_data` 函数

在程序最后调用。用与先前的程序同样的方法，将运行时的日期和时间写入 "`datetime.dat`" 文件。

● 练习 13-5

在代码清单 13-4 的程序中加上表示当前“心情”的字符串。即在显示上一次的运行时间（和上一次的心情）之后提示输入“当前的心情：”，从键盘读入字符串再写入文件。例如，如果输入“极好！！”，那么程序在下一次运行时就应显示“上一次运行是在 XXXX 年 XX 月 XX 日 XX 时 XX 分 XX 秒，心情极好！！”。

代码清单 13-4

```

/*
  显示程序上一次运行时的日期和时间
*/

#include <time.h>
#include <stdio.h>

char data_file[] = "datetime.dat";          /* 文件名 */

/*--- 取得并显示上一次运行时的日期和时间 ---*/
void get_data(void)
{
    FILE *fp;

    if ((fp = fopen(data_file, "r")) == NULL) {          /* 打开文件 */
        printf("本程序第一次运行。\\n");
    } else {
        int year, month, day, h, m, s;

        fscanf(fp, "%d%d%d%d%d", &year, &month, &day, &h, &m, &s);
        printf("上一次运行是在%d年%d月%d日%d时%d分%d秒。\\n",
               year, month, day, h, m, s);
        fclose(fp);          /* 关闭文件 */
    }
}

/*--- 写入本次运行时的日期和时间 ---*/
void put_data(void)
{
    FILE *fp;
    time_t t;
    struct tm *local;

    time(&t);
    local = localtime(&t);

    if ((fp = fopen(data_file, "w")) == NULL)          /* 打开文件 */
        printf("\\a文件打开失败。\\n");
    else {
        fprintf(fp, "%d %d %d %d %d %d\\n",
                local->tm_year + 1900, local->tm_mon + 1, local->tm_mday,
                local->tm_hour, local->tm_min, local->tm_sec);
        fclose(fp);          /* 关闭文件 */
    }
}

int main(void)
{
    get_data();          /* 取得并显示上一次运行时的日期和时间 */

    put_data();          /* 写入本次运行时的日期和时间 */

    return (0);
}

```

标准输入输出

第 8 章中曾出现过代码清单 13-5 中的程序。

该程序的功能是将从标准输入流（一般为键盘）输入的字符复制到标准输出流（一般为显示器界面）。大家想起来了吗？

代码清单 13-5

```
/*
 * 将来自标准输入的输入复制到标准输出
 */
#include <stdio.h>

int main(void)
{
    int ch;

    while ((ch = getchar()) != EOF)
        putchar(ch);

    return (0);
}
```

显示文件内容

如果程序不从标准输入流读取数据，而是从任意文件读取，那就变成更实用的查看文件内容（在界面上显示）的程序了。请看代码清单 13-6。

程序首先提示输入文件名，将文件名读入字符串 *fname*。如果文件打开失败，就会显示“文件打开失败”，这和之前的程序相同。

作为本程序主体的 **while** 语句和代码清单 13-5 如出一辙。区别仅在于将 **getchar()** 的调用改成了 **fgetc(*fp*)**。**fgetc** 函数说明如下：

fgetc	
头文件	#include <stdio.h>
原 型	int fgetc(FILE *stream)
说 明	从stream指向的输入流（若存在）中读取unsigned char型的下一个字符的值，并将它转换为int型。然后，若定义了流的文件位置指示符，则将其向前移动。
返回值	返回stream所指输入流中的下一个字符。若在流中检查到文件末尾，则设置该流的文件结束指示符并返回EOF。如果发生读取错误，就设置该流的错误指示符并返回EOF。

代码清单 13-6

```

/*
    显示文件内容
*/
#include <stdio.h>

int main(void)
{
    int ch;
    FILE *fp;
    char fname[64];          /* 文件名 */

    printf("文件名: ");
    scanf("%s", fname);

    if ((fp = fopen(fname, "r")) == NULL)    /* 打开文件 */
        printf ("\a文件打开失败.\n");
    else {
        while ((ch = fgetc(fp)) != EOF)
            putchar(ch);
        fclose(fp);                /* 关闭文件 */
    }

    return (0);
}

```

运行结果

```

文件名: list1306.c
/*
    显示文件内容
*/
#include <stdio.h>

(以下省略)

```

与 **getchar** 函数相比，仅增加了一个参数，即输入流。

当从文件正常读取到字符时，就会进入 **while** 循环语句，通过下述语句将读取到的字符 *ch* 显示界面上。

```
putchar(ch);
```

当达到文件末尾（后面没有字符）或者有错误发生时，就会结束 **while** 语句循环并关闭文件，程序结束运行。

● 练习 13-6

编写程序实现从键盘读入文件名，计算该文件的行数（换行符的个数）并显示在界面上。

● 练习 13-7

编写程序实现从键盘读入文件名，计算该文件的字符数并显示在界面上。

文件的复制

如果将从文件读取到的字符输出到任意文件，而不是输出到标准输出流，那就变成更为实用的文件复制程序了。

请看代码清单 13-7 所示的程序。

代码清单 13-7

```
/*
 复制文件
*/

#include <stdio.h>

int main(void)
{
    int    ch;
    FILE   *sfp, *dfp;
    char   sname[64], dname[64];          /* 文件名 */

    printf("打开源文件: ");      scanf("%s", sname);
    printf("打开目标文件: ");    scanf("%s", dname);

    if ((sfp = fopen(sname, "r")) == NULL)          /* 打开源文件 */
        printf("\a源文件打开失败.\n");
    else {
        if ((dfp = fopen(dname, "w")) == NULL)      /* 打开目标文件 */
            printf("\a目标文件打开失败.\n");
        else {
            while ((ch = fgetc(sfp)) != EOF)
                putc(ch, dfp);
            fclose(dfp);                             /* 关闭目标文件 */
        }
        fclose(sfp);                                 /* 关闭源文件 */
    }

    return (0);
}
```

► 省略运行结果。

这个程序涉及两个文件的操作，较之前面的程序稍显复杂。

程序首先会询问需要复制的“源文件”和“目标文件”的文件名，并将它们读入字符串 *sname* 和 *dname*。

然后以只读模式打开源文件，并将指向该文件相关联的流的指针赋给 *sfp*。

如果文件打开成功，就以只写模式打开目标文件，并将指向该文件相关联的流的指针赋给 *dfp*。

如果两个文件都打开成功，就运行本程序的主体 **while** 语句。

while 语句和前面的程序类似，只是将 **putchar()** 改成了 **fputc(*ch*, *dfp*)**。**fputc** 函数的说明如下：

fputc	
头文件	#include <stdio.h>
原型	int fputc(int <i>c</i>, FILE *<i>stream</i>);
说明	将 <i>c</i> 指定的字符转换为 unsigned char 型后写入 <i>stream</i> 指向的输入流。此时如果定义了流的文件位置指示符，就会向指示符指向的位置写入字符，并将文件位置指示符适当地向前移动。在不支持文件定位或者以追加模式打开流的情况下，总是以向输出流的末尾追加字符的方式进行字符输出。
返回值	返回写入的字符。如果发生写入错误，就设置该流的错误指示符并返回 EOF。

与 **putchar** 函数相比，仅增加了第二个参数，即输出流。

当从文件读入字符时，会进入 **while** 循环语句，通过下述语句将读入的字符 *ch* 输出至流 *dfp*。

```
fputc (ch, dfp);
```

当遇到文件末尾（后面没有字符）或者有错误发生时，就会结束循环并关闭文件，程序结束运行。

至此，文件复制完成。

● 练习 13-8

请参考代码清单 13-7 编写一个程序，在界面上显示文件内容的同时执行复制操作。

● 练习 13-9

请参考代码清单 13-7 编写一个程序，将所有英文小写字母转换为大写字母的同时执行复制操作。

● 练习 13-10

请参考代码清单 13-7 编写一个程序，将所有英文大写字母转换为小写字母的同时执行复制操作。

13-2 文本和二进制

在文本文件中保存实数

请看代码清单 13-8 所示的程序。程序先将初始化为圆周率 3.14159265358979323846 的变量 *pi* 的值写出至 "PI.txt" 文件，然后再进行读取和显示。

代码清单 13-8

```
/*
 将圆周率的值写入文本文件，再进行读取
*/
#include <stdio.h>

int main(void)
{
    FILE    *fp;
    double  pi = 3.14159265358979323846;

    /* 写入操作 */
    if ((fp = fopen("PI.txt", "w")) == NULL) /* 打开文件 */
        printf("\a文件打开失败.\n");
    else {
        fprintf(fp, "%f", pi);                /* 从pi写入 */
        fclose(fp);                          /* 关闭文件 */
    }

    /* 读取操作 */
    if ((fp = fopen("PI.txt", "r")) == NULL) /* 打开文件 */
        printf("\a文件打开失败.\n");
    else {
        fscanf(fp, "%lf", &pi);               /* 读取至pi */
        printf("圆周率是%23.31f.\n", pi);     /* 关闭文件 */
        fclose(fp);
    }

    return (0);
}
```

运行结果示例

圆周率是 3.141592999999999858000

运行结果比较奇怪，3.141592 为止都是正确的，之后的部分就不对了。通过编辑器查看由这个程序建立的 "PI.txt" 文件，可以发现里面的内容是 3.141593（见图 13-7）。

由于调用 **fprintf** 函数时未指定精度，浮点数默认只输出小数点后 6 位数字，所以产生了这样的输出结果（**printf** 函数同样如此）。

3.141593

图 13-7 "PI.txt" 文件内容

我们无法通过这个数据恢复丢失的部分。

► **fscanf** 函数将从文件读取到的 3.141593 保存至变量 *pi*。由于 **double** 型并不能毫无误差地显示实数的所有位数，所以在 **printf** 函数中指定显示小数点后 21 位时，无法保证恰好显示 3.14159300000000000000。

要做到不丢失任何一位数据，就必须写入所有位数。所以我们要注意向文件写入时的精度（位数），写出的字符数（位数）可能会相应地增大。

文本文件和二进制文件

我们用二进制文件来解决这个问题。首先要明确文本文件和二进制文件的区别。

■ 文本文件

在文本文件中，数据是以字符序列的形式表示的。例如，整数 357 是 '3'、'5'、'7' 三个字符的序列。若使用 **printf** 函数和 **fprintf** 函数将值写入控制台界面或文件，则会占去 3 个字节。同理，如果是数值 2057 的话，就会写出 '2'、'0'、'5'、'7' 四个字符。

如果字符编码是 ASCII 码，那么这些数值数据就会由图 13-8(a) 所示的二进制位构成。

由此可见，文本文件的字符数取决于数值位数。

■ 二进制文件

在二进制文件中，数据是以二进制位串的形式表示的。具体位数虽因编译器而异，但 **int** 型整数的长度必定为 **sizeof(int)** 的值。

如果是用 2 个字节（16 位）表示 **int** 型整数的环境，那么整数 357 和 2057 就将由图 13-8(b) 所示的二进制位构成。

由此可见，二进制文件的字符数（字节数）不依赖于数值位数。

(a) 文本 长度（字符数）要和位数相同

整数值 357	00110011	00110101	00110111
整数值 2057	00110010	00110000	00110101 00110111

(b) 二进制 长度固定为 **sizeof(int)**

整数值 357	0000000101100101
整数值 2057	0000100000001001

图 13-8 文本和二进制

在二进制文件中保存实数

代码清单 13-9 所示的程序改用二进制模式对圆周率的值进行读写。

fwrite 函数和 **fread** 函数分别用于二进制文件的写入和读取。以下是函数说明。

fwrite	
头文件	#include <stdio.h>
原 型	size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
说 明	从ptr指向的数组中将最多nmemb个长度为size的元素写入stream指向的流中。若定义了流的文件位置指示符，则以成功写入的字符数为单位向前移动。当发生错误时，该流的文件位置指示符的值不可预测。
返回值	返回成功写入的元素个数。仅当发生写入错误时，元素个数会少于nmemb。

fread	
头文件	#include <stdio.h>
原 型	size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
说 明	从stream指向的流中最多读取nmemb个长度为size的元素到ptr指向的数组。若定义了流的文件位置指示符，则以成功读取的字符数为单位向前移动。当发生错误时，该流的文件位置指示符的值不可预测。只读取到某一元素的部分内容时，值不可预测。
返回值	返回成功读取的元素个数。当发生读取错误或达到文件末尾时，元素个数会少于nmemb。若size或nmemb为0，则返回0。这时数组内容和流的状态都不发生变化。

这两个函数会接收 4 个参数。第一个参数是指向读写数据的首地址的指针，第二个参数是数据的长度，第三个参数是数据的个数，第四个参数是指向读写对象的流的指针。

在本程序中，向文件写入的函数是：

```
fwrite(&pi, sizeof(double), 1, fp);          /* 从pi写入 */
```

从文件读取的函数是：

```
fread(&pi, sizeof(double), 1, fp);          /* 读取至pi */
```

第二个参数 **sizeof(double)** 指定了 **double** 型的长度，第三个参数指定了要读写的变量个数为 1 个。

► **sizeof**（数据类型名称）是取得该数据类型长度的运算符。

这两个函数是为一次完成对数组元素（不是单独的变量）的读写而设计的。如果读写对象不是数组，而是单独的变量，那么函数的调用方式将和读写数组的典型调用方式有所不同。表 13-1 中对这两者进行了对比。

代码清单 13-9

```
/*
将圆周率的值写入二进制文件再进行读取
*/

#include <stdio.h>

int main(void)
{
    FILE    *fp;
    double  pi = 3.14159265358979323846;

    /* 写入操作 */
    if ((fp = fopen("PI.bin", "wb")) == NULL) /* 打开文件 */
        printf("\a 文件打开失败.\n");
    else {
        fwrite(&pi, sizeof(double), 1, fp); /* 从 pi 写入 */
        fclose(fp);                          /* 关闭文件 */
    }

    /* 读取操作 */
    if ((fp = fopen("PI.bin", "rb")) == NULL) /* 打开文件 */
        printf("\a 文件打开失败.\n");
    else {
        fread(&pi, sizeof(double), 1, fp); /* 读取至 pi */
        printf(" 圆周率是 %23.31f.\n", pi);
        fclose(fp);                        /* 关闭文件 */
    }

    return (0);
}
```

运行结果示例

圆周率是 3.141592653589793116000。

表 13-1 fwrite 函数和 fread 函数的典型用例

	int x;	int a[10];
写入操作	fwrite(&x, sizeof(int), 1, fp);	fwrite(a, sizeof(int), 10, fp);
读取操作	fread(&x, sizeof(int), 1, fp);	fread(a, sizeof(int), 10, fp);

● 练习 13-11

编写一个程序，读取含有 10 个 double 型元素的数组的所有元素值。

● 练习 13-12

改写代码清单 13-4 的程序，将日期和时间作为 struct tm 型的值直接向二进制文件进行读写操作。

显示文件自身

代码清单 13-6 所示的“查看文件内容”的程序是以文本文件为对象的。因此，如果用这个程序查看包含非打印字符的二进制文件，那么输出内容看起来就会比较混乱，不能正确显示。

代码清单 13-10

```

/*
 用字符和字符编码显示文件内容
*/

#include <ctype.h>
#include <stdio.h>

int main(void)
{
    int          n;
    unsigned long count = 0;
    unsigned char buf[16];
    FILE *fp;
    char  fname[64];          /* 文件名 */

    printf("文件名: ");
    scanf("%s", fname);

    if ((fp = fopen(fname, "rb")) == NULL)          /* 打开文件 */
        printf("\a 文件打开失败。\\n");
    else {
        while ((n = fread(buf, 1, 16, fp)) > 0) {
            int i;

            printf("%08lx ", count);                /* 地址 */

            for (i = 0; i < n; i++)                  /* 十六进制 */
                printf("%02X ", (unsigned)buf[i]);

            if (n < 16)
                for (i = n; i < 16; i++) printf(" ");

            for (i = 0; i < n; i++)                  /* 字符 */
                putchar(isprint(buf[i]) ? buf[i] : '.');

            putchar('\\n');

            count += 16;
        }
        fclose(fp);                                /* 关闭文件 */
    }

    return (0);
}

```

代码清单 13-10 所示的程序分别用“字符”和“字符编码（十六进制）”显示了以二进制文件类型打开的文件内容。

在显示“字符”时，使用了 **isprint** 函数对字符进行判断，如果是打印字符就显示该字符，如果不是打印字符就用 '.' 代替。

isprint	
头文件	#include <ctype.h>
原 型	int isprint(int c);
说 明	判断字符c是否为可打印字符（含空格）。
返回值	若判断成功则返回0以外的值（真），否则返回0。

这个程序显示了代码清单 13-10 自身的内容（见图 13-9）。

► 图中显示的运行结果仅供参考。实际运行结果取决于程序运行环境所使用的字符编码。

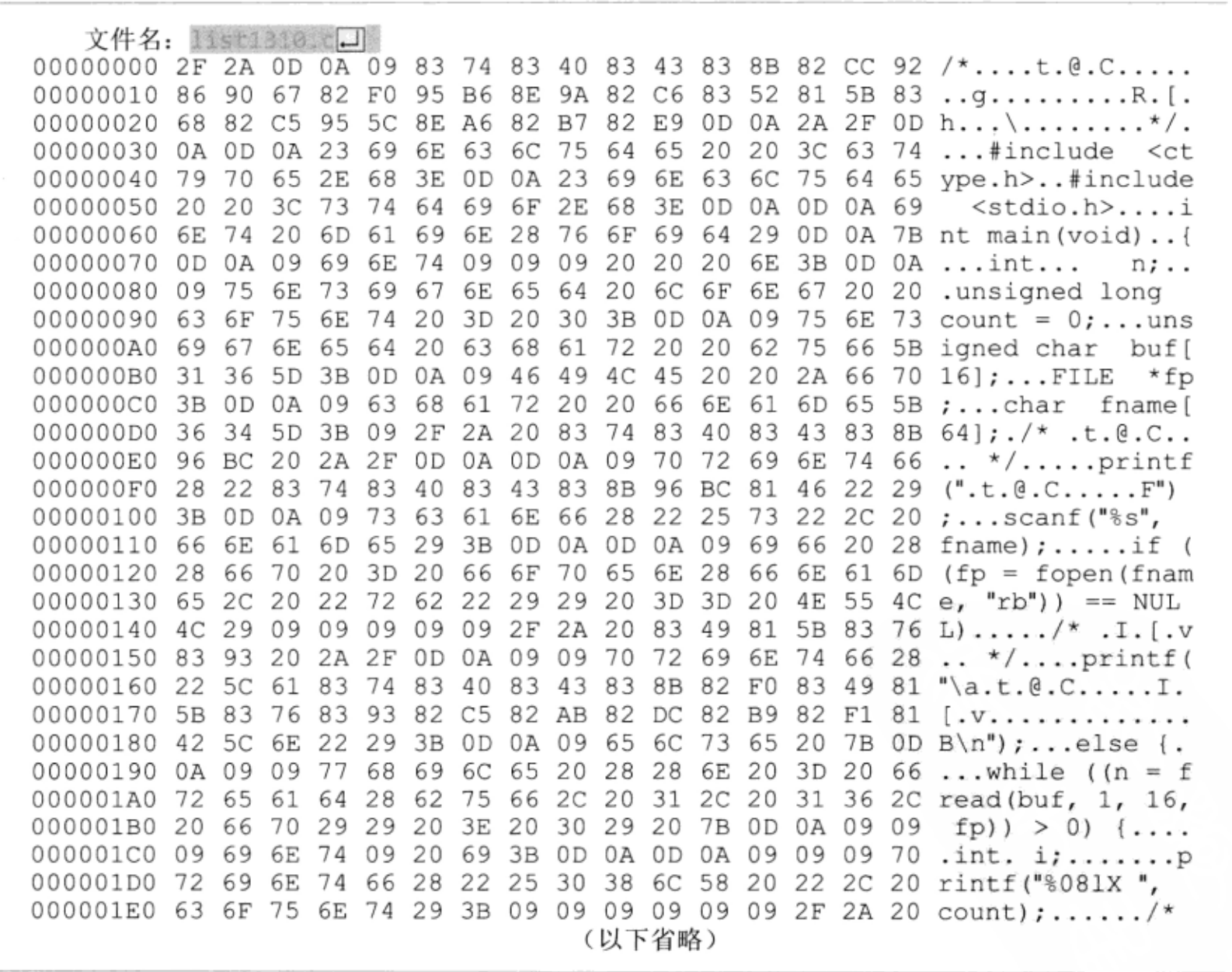


图 13-9 代码清单 13-10 的运行结果示例



附录1

C语言简介

本篇将对 C 语言的历史背景作一个简单介绍。



C 语言的历史

C 语言的前身是 Martin Richards 开发的 BCPL 语言。1970 年 Ken Thompson 对 BCPL 语言进行了改进，发明了 B 语言。

1972 年 Dennis M.Ritchie 又在 B 语言的基础上开发出了 C 语言。

当时，Ritchie 和 Ken Thompson 等人一同致力于小型机操作系统 UNIX 的开发。UNIX 操作系统最初是用汇编语言开发的，之后用 C 语言进行了重写。

C 语言是为了移植早期的 UNIX 而开发出来的，所以从某种意义上说“C 语言是 UNIX 的副产品”。

不只是 UNIX 本身，就连运行在 UNIX 系统上的许多应用程序也是接二连三地使用 C 语言开发出来的。

因此，C 语言首先普遍应用于 UNIX 世界，接着又凭借其势不可挡的魅力在大型计算机和个人计算机领域得到了广泛普及。

K&R——C 语言的圣经

Ritchie 与 Brian W.Kernighan 合著了一本 C 语言教材：

The C Programming Language, Prentice-Hall, 1978（中文版名为《C 程序设计语言》）

这是 C 语言设计者亲自撰写的书，被众人奉为 C 语言的“圣经”，热心的读者们还结合两位作者的姓氏首字母“K&R”，将其作为书的昵称。

在 K&R 的附录部分收录了 C 语言规范的参考手册（Reference Manual）。这个语言规范被认为是 C 语言的标准。

C 语言标准规范

K&R 的参考手册中规定的 C 语言规范还存在着不少未完全明确的部分。而且，随着 C 语言的普及，衍生出了许多“方言”，这些各自拥有扩展功能的 C 语言随处可见。

原本 C 语言的优势就在于可移植性强，能方便地将一种计算机平台上开发的 C 语言程序移植到另一种平台上运行。但是由于这些方言的影响，可移植性逐渐下降。

这时制定 C 语言国际标准的活动便应运而生。由于关系到全球 C 语言标准的统一，此项工作是在非常严谨的过程中展开的。

国际标准化组织 ISO (International Organization for Standardization) 和美国国家标准学会 ANST (American National Standards Institute) 通力合作完成了这项艰巨的工作。

1989 年 12 月，首先制定了美国国家标准：^①

ANSI X3.159-1989

American National Standard for Information Systems - Programming Language-C

1990 年 12 月，制定了国际标准：^②

INTERNATIONAL STANDARD ISO/IEC 9899 : 1990(E) Programming Languages-C

这两个标准体裁各异，但内容完全一致。

1993 年，日本也制定出了相同内容的标准。^③

JIS X3010-1993 程序设计语言 C

*

也许是因为 ANSI 标准比 ISO 标准制定得早，加上 ANSI 在日本的知名度较高，很多人将遵循标准规范的 C 语言叫作“ANSI C”。

但是 ANSI 是美国标准，在 ISO 国际标准和 JIS 日本标准中有着同样的规范，因此应该将它称为“标准 C”，而不是 ANSI C。

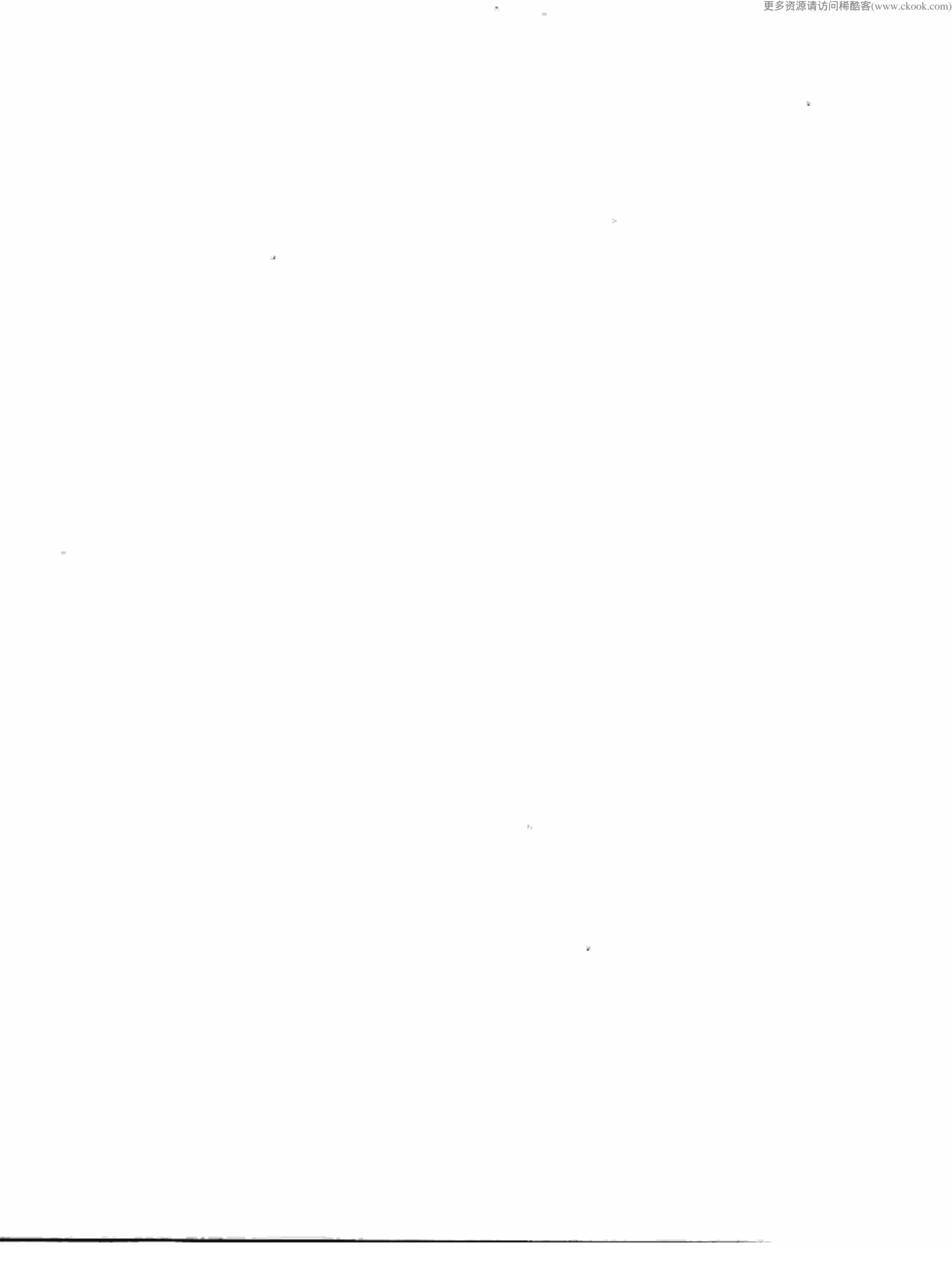
► 此后对标准 C 的规范进行了修订，加入了可变长数组、`long long int` 型，取消了不写函数返回类型默认就是 `int` 型的规定，扩充了数学函数库（其中包括增加对复数运算的支持）等。修订后的标准为 ISO、ANSI、JIS 的“第 2 版”。由于该标准制定于 1999 年，所以称为“C99”。

但目前几乎没有一个编译器完全支持新标准，该标准没有得到广泛应用（有望在将来逐步推广）。

① 该标准制定于 1989 年，所以也称为“C89”标准。

② 该标准制定于 1990 年，所以也称为“C90”标准。

③ 1994 年 12 月，中国发布了程序设计语言 C 的国家标准（标准编号：GB/T 15272-1994）采用的即是上述“C90”标准。参考自中国标准网 <http://www.chinaios.com/>。



附录2

printf函数与scanf函数

本篇将对 **printf** 函数与 **scanf** 函数进行详细说明。



printf 函数

原型

```
#include <stdio.h>

int printf(const char *format, ...);
```

► ...称为省略号 (ellipsis)。表示可以接受任意个数的形参。

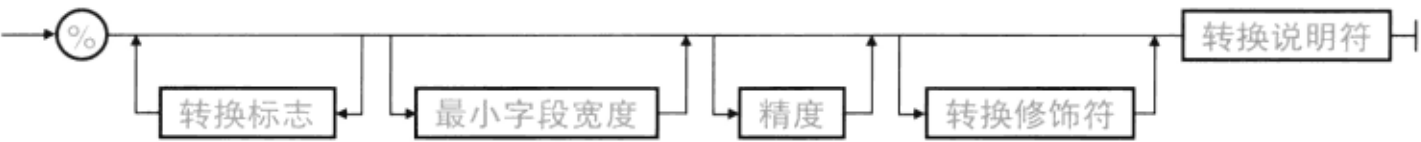
功能

printf 函数会将 format 后面的实参转换为指定的字符序列输出形式，再将它发送至标准输出流。这个转换是根据 format 所指的格式控制字符串中的命令进行的。格式控制字符串中可以不包含任何命令，也可以包含多个命令。

► 实参个数比格式控制字符串少时的操作未定义。实参个数比格式控制字符串多时，多余的实参将被忽略。

命令可分为下述两类：

- ◆ % 以外的字符，不作转换按原样复制到输出流。
- ◆ 转换说明，对后面给出的 0 个以上的实参作格式转换（语法见下图）。



转换标志

使用标志字符 -、+、空格、#、0 可以修饰转换说明的含义。可指定 0 个以上（包括 0）的标志，顺序任意。

-	将转换结果在字段宽度范围内左对齐。未指定时右对齐。
+	总是要转换的带符号数值之前加上正号或负号。未指定时只对负值加负号。
空格	若带符号的转换结果不以符号开头或者字符数为0，则在数值前加上空格。 ► 若同时指定了空格标志和 + 标志，则空格标志无效。
#	为以下数值表示形式（基数等）作格式转换。 <ul style="list-style-type: none">• o 转换——第一个数字为 0（增加精度）。• x、X 转换——在数值之前加上前缀 0x(或 0X)。数值为 0 时不加前缀。• e、E、f、g、G 转换——无论小数点之后是否有数字，都加上小数点（一般只在小数点后有数字的情况下才加）。• g、G 转换——保留转换结果末尾的 0。• 其他转换操作未定义。

(续)

0	<ul style="list-style-type: none">• d、i、o、u、x、X、e、E、f、g、G 转换——在字段宽度范围的左侧使用 0 而非空格进行填充（符号和基数位于 0 的前面）。• 其他转换操作未定义。<ul style="list-style-type: none">▶ 若同时指定 0 标志和 - 标志，则 0 标志无效。▶ 若在 d、i、o、u、x、X 转换中指定了精度，则 0 标志无效。
---	--

最小字段宽度

可以用 “*” 或十进制整数表示。

若转换结果的字符数小于指定的最小字段宽度，则在左侧(指定 - 标志时在右侧)补空格(若未指定 0 标志)，直到填满字段宽度。

精度

可以用小数点 . 后的星号 * 或十进制整数表示。省略十进制整数时精度为 0。对于各种转换的说明如下：

- d、i、o、u、x、X 转换——最小输出位数。
- e、E、f 转换——小数点之后的输出位数。
- g、G 转换——最大有效位数。
- s 转换——最大字符数。
- 其他转换操作未定义。

▶ 当用星号指定字段宽度、精度时，需要有相应的 int 型实参（定义在要转换的实参之前）。当指定了字段宽度的实参为负数时，则字段宽度解释为前置 - 标志的正数。当指定了精度的实参为负数时，则精度解释为未指定。

转换修饰符

可以用 h、l、L 表示，可缺省。

h	<ul style="list-style-type: none">• d、i、o、u、x、X 转换——表示实参的数据类型为 short 型或 unsigned short 型（实参会根据数据类型自动提升规则转换成高精度的数值进行计算。计算完成后再将值转回 short 型或 unsigned short 型进行显示）。• n 转换——表示实参的数据类型为指向 short 型的指针。
l	<ul style="list-style-type: none">• d、i、o、u、x、X 转换——表示实参的数据类型为 long 型或 unsigned long 型。• n 转换——表示实参的数据类型为指向 long 型的指针。
L	<ul style="list-style-type: none">• e、E、f、g、G 转换——表示实参的数据类型为 long double 型。

转换修饰符与上述以外的其他转换说明符一起使用时的操作未定义。

转换说明符

可以用 **d**、**i**、**o**、**u**、**x**、**X**、**f**、**e**、**E**、**g**、**G**、**c**、**s**、**p**、**n**、**%** 表示。

d、**i** 将 **int** 型的实参转换为 `[-] dddd` 形式的带符号十进制数进行输出。

精度指定了应输出的数字的最少个数。若转换结果的数字个数（位数）小于指定的精度，则在前面补 0 直到满足精度要求。缺省时的精度默认为 1。精度为 0 且参数为 0 时，转换结果的位数为 0（null 字符串）。

o、**u**、**x**、**X** 将 **unsigned** 型的实参转换为 `dddd` 形式的无符号八进制数（**o**）、无符号十进制数（**u**）、无符号十六进制数（**x** 或 **X**）。在 **x** 转换中，使用字符 `abcdef`。在 **X** 转换中，使用字符 `ABCDEF`。

精度指定了应输出的最少位数。若转换结果的位数小于指定的精度，则在前面补 0 直到满足精度要求。缺省时的精度默认为 1。精度为 0 且参数为 0 时，转换结果的位数为 0（null 字符串）。

f 将 **double** 型的实参转换为 `[-] ddd.ddd` 形式的十进制数进行输出。

此时小数点之后的位数等于指定的精度。缺省时的精度默认为 6。如果精度为 0 且未指定 **#** 标志，则不会输出小数点。小数点之前至少有 1 个数字时才会输出小数点。

该转换会根据位数适当地四舍五入。

e、**E** 将 **double** 型的实参转换为 `[-] d.ddde ± dd` 形式的十进制数进行输出。

此时小数点之前输出 1 位（实参为 0 时除外，不为 0 的）数字，小数点之后输出与指定精度相同位数的数字。缺省时的精度默认为 6。如果精度为 0 且未指定 **#** 标志，则不会输出小数点。该转换会根据位数适当地四舍五入。指定 **E** 转换时，指数前的字符是 **E** 而不是 **e**。

指数总是至少显示 2 位。值为 0 时，指数的值为 0。

g、**G** 根据指定了有效位数的精度，将 **double** 型的实参转换为 **f** 形式或 **e** 形式（指定 **G** 转换时为 **E** 形式）。

精度为 0 时，解释为 1。

使用哪种形式取决于待转换的值。若转换结果中的指数小于 -4 或大于等于精度，则使用 **e** 形式（或 **E** 形式）。无论使用哪种形式，都会去掉转换结果小数部分缀尾的 0。只有当小数点后还有数字的情况下，才会输出小数点。

c 将 **int** 型的实参转换为 **unsigned char** 型之后，再输出转换后的字符。

s 实参必须为指向字符型数组的指针。输出数组中 `null` 之前的字符。当指定了精度时，不会输出超出精度范围的字符。当精度未指定或精度大于数组长度时，数组必须包含 `null` 字符。

P 实参必须为指向 **void** 的指针。用编译器所定义的格式将该指针的值转换为可显示的字符序列。

n 实参必须为指向整数的指针。这个整数保存了到调用该 **printf** 函数为止发送至输出流的字符数。不进行实参的转换。

% 输出 %。无实参。转换说明必须写作 `%%`。

指定无效的转换说明符时的操作未定义。

当实参为共用体或聚合体，抑或是指向这两者的指针时(除了 `%s` 转换时的字符型数组和 `%p` 转换时的指针)，操作未定义。

字段宽度未指定或比转换结果的长度小时，不会截断转换结果。即当转换结果的字符数大于字段宽度时，将宽度扩大至正好能容纳转换结果。

■ 返回值

printf 函数会返回输出的字符数。发生输出错误时，返回负值。

scanf 函数

原型

```
#include <stdio.h>

int scanf(const char *format, ...);
```

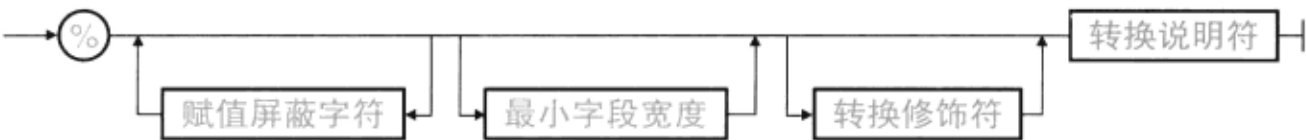
功能

scanf 函数的功能是对来自于标准输入流的输入数据作格式转换，并将转换结果保存至 *format* 后面的实参所指向的对象。*format* 指向的字符串为格式控制字符串，它指定了可输入的字符串及其赋值时转换方法。格式控制字符串中可以不包含任何命令，也可包含多个命令。

- ▶ 实参个数比格式控制字符串少时的操作未定义。
- 实参个数比格式控制字符串多时，多余的实参将被忽略。

命令可分为下述三类：

- ◆ 1 个以上的空白字符。
- ◆ (%) 和空白字符以外的) 字符。
- ◆ 转换说明（语法见下图）。



赋值屏蔽字符

用 “*” 表示。可缺省。

最大字段宽度

用 0 以外的十进制整数表示。可缺省。

转换修饰符

表示保存转换结果的对象的长度。可以用 **h**、**l**、**L** 表示。

h	<ul style="list-style-type: none">• d、i、n 转换——表示实参为指向 short 型的指针，而不是指向 int 型的指针。• o、u、x 转换——表示实参为指向 unsigned short 型的指针，而不是指向 unsigned 型的指针。
---	---

(续)

l	<ul style="list-style-type: none">• d、i、n 转换——表示实参为指向 long 型的指针，而不是指向 int 型的指针。• o、u、x 转换——表示实参为指向 unsigned long 型的指针，而不是指向 unsigned 型的指针。• e、f、g 转换——表示实参为指向 double 型的指针，而不是指向 float 型的指针。
L	<ul style="list-style-type: none">• e、f、g 转换——表示实参为指向 long double 型的指针，而不是指向 float 型的指针。

转换修饰符与上述以外的其他转换说明符一起使用时的操作未定义。

scanf 函数会依次执行格式控制字符串中的各项命令。命令执行失败时，**scanf** 函数会返回主调函数。以下两个原因会导致命令执行失败：

- (a) **输入错误**——由于获取不到输入字符而导致。
- (b) **匹配错误**——由于不恰当的输入而导致。

由空白字符构成的命令会读取输入的空白字符，直至出现第一个非空白字符（该字符不会被读取，保留在流中）或者不能继续读取为止。命令通常会从流中读取下一个字符。当输入字符与构成命令的字符不匹配时，这项命令失败并且该输入字符及其后面的字符都不会被读取，仍然保留在流中。

转换说明的命令根据各个转换说明符相应的规则定义输入匹配项的集合。转换说明按下述步骤执行。

若转换说明中不包含 **l**、**c**、**n** 指定符，则在读取时会跳过空白字符串。若转换说明中不包含 **n** 指定符，则会从流中读取输入项。输入项为输入字符串中最长的匹配项。但如果最长的匹配项的长度超过了指定的字段宽度，就截取匹配项中与字符宽度相等的前几个字符作为输入项。即使在输入项后面还有字符也不会被读取，该字符及其后面的字符都将保留在流中。当输入项的长度为 0 时，命令执行失败，此时状态为匹配错误。而因某些错误导致不能在流中进行输入，则为输入错误。

除非使用 **%** 指定符，其它情况下转换说明都会根据转换说明符将输入项（或 **%n** 指定时的输入字符数）转换为合适的数据类型。当输入项不为匹配项时，命令执行失败，此时状态为匹配错误。如果没有指定输入屏蔽字符 *****，那就会将转换结果赋于 *format* 之后还未赋值的第一个实参指向的对象中。该对象的数据类型不正确或不能在内存单元中表示转换结果时的操作未定义。

转换说明符

可以用 `d`、`i`、`o`、`u`、`x`、`X`、`e`、`E`、`f`、`g`、`G`、`s`、`[]`、`c`、`p`、`n`、`%` 表示。

`d` 可省略符号十进制整数。实参必须为指向整数的指针。

`i` 可省略符号的整数。实参必须为指向整数的指针。

`o` 可省略符号的八进制整数。实参必须为指向无符号整数的指针。

`u` 可省略符号的十进制整数。实参必须为指向无符号整数的指针。

`x`、`X` 可省略符号的十六进制整数。实参必须为指向无符号整数的指针。

`e`、`E`、`f`、`g`、`G` 可省略符号的浮点数。实参必须为指向浮点数的指针。

`s` 非空白字符序列。实参必须为指向字符型数组第一个字符的指针。该数组的长度须足够容纳所有字符序列以及 `null` 字符。该转换会自动添加一个表示字符串末尾的 `null` 字符。

`[]` 扫描字符集 (`scanset`) 元素的非空序列。实参必须为指向字符型数组第一个字符的指针。该数组的长度须足够容纳所有字符序列以及 `null` 字符。该转换会自动添加一个表示字符串末尾的 `null` 字符。

转换说明符为左方括号与右方括号 “`]`” 之间的格式控制字符串中的所有字符序列。当紧跟在左方括号后面的字符不是折音符 `^` 时，扫描字符集由左右方括号之间的**扫描列表** (`scanlist`) 构成。当紧跟在左方括号后面的字符是 `^` 时，扫描集为未出现在 `^` 与右方括号之间的扫描列表中的所有字符。当转换说明符以 `[]` 或字符开始时，第一个右方括号为扫描列表中的一个字符元素，而第二个出现的右方括号才是转换说明的结束符。当转换说明符不以 `[]` 和 `[^]` 开始时，第一个出现的右方括号就是转换规范的结束符。当扫描列表中含有连字符 `-`，并且既非第一个字符（如果以 `^` 开头，则为第二个字符）也非最后一个字符时，其定义因编译器而异。

`c` 字符宽度（命令中没有指定字段宽度时默认为 1）中指定长度的字符序列。该指定符对应的实参必须为指向字符型数组的第一个字符的指针。该数组的长度须能足够容纳接收到的字符序列。该转换不会添加 `null` 字符。

p 编译器定义的字符序列的集合。该集合与 **printf** 函数中的 **%p** 转换所生成的字符序列集合相同。该指定符对应的实参必须为指向 **void** 的指针的指针。对输入项的解释根据编译器而定。如果输入项为同一程序中已转换过的值，那么转换结果的指针值与转换前的值相等。其他情况时的 **%p** 转换操作未定义。

n 不读取输入。该指定符对应的实参必须为指向整数的指针。这个整数保存了到调用 **scanf** 函数为止从输入流读取到的字符数。执行 **%n** 命令并不会增加 **scanf** 函数结束时返回的输入项数。

% 匹配一个 **%**。不会执行转换和赋值操作。转换说明必须写作 **%%**。

指定无效的转换说明符时的操作未定义。

如果在输入中检测到文件末尾就结束转换操作。如果在检测到文件末尾之前，未读取到任何 1 个字符匹配当前命令，那么就视该命令在执行中发生输入错误，结束转换操作。如果在检测到文件末尾之前，至少读取到 1 个字符匹配当前命令，那么只要该命令不发生匹配错误，后续命令（若存在）就会因发生输入错误而结束操作。

若因输入字符与命令不匹配使得转换操作结束，那么这个不匹配的输入字符就不会被读取，仍然保留在流中。只要输入中后续的空白字符（包括换行符）与命令不匹配，就会保留在流中不被读取。除非使用 **%n** 命令，通常字符命令以及包含赋值屏蔽的转换规范都无法直接判断执行是否成功。

■ 返回值

如果不作任何转换就发生了输入错误，**scanf** 函数会返回宏定义 **EOF** 的值。否则，**scanf** 函数会返回成功赋值的输入项数。如果输入时发生了匹配错误，那么这个项数就会比转换说明符对应的实参个数少，甚至为 0。

致谢

在编写本书之际，有幸得到了 Softbank Creative 有限公司野泽美西男主编的大力支持和帮助。

出色的计算机技术讲师赤尾浩氏老师也在百忙之中欣然接受了本书原稿的审校工作。

我想借此机会，对两位深表谢意！

（图灵公司感谢臧秀涛针对 C99 标准做出的译者注，感谢臧秀涛、李琳骁、白顺龙、朱伟杰、邓国平、何文祥对本书的审读）

参考文献

- 1) Brian W. Kernighan and Dennis M. Ritchie

The C Programming Language Second Edition, Prentice Hall, 1988

- 2) American National Standards Institute

ANSI/ISO 9899-1990 American National Standard for Programming Languages - C, 1992

- 3) 日本工業規格

JIS X3010-1993, プログラミング言語 C, 1993

- 4) 平林雅英

ANSI C/C++辞典, 共立出版社, 1996

- 5) 柴田望洋

秘伝 C 言語問答ポイント編, ソフトバンク, 1991

- 6) 柴田望洋

Dr.望洋のプログラミング道場, ソフトバンク, 1993

- 7) 柴田望洋

プログラミング講義C++, ソフトバンク, 1996

索引

符号

, (实参分隔符)	4	-> 运算符	276
, (逗号运算符)	110, 189	- 标志	318
, (初始值分隔符)	103	= (基本赋值运算符)	11, 23
, (声明分隔符)	11	== 运算符	42
... (省略号)	318	<= 运算符	44
. 运算符	274	<<= 运算符	66
; (声明结束符)	10	<< 运算符	166
; (语句结束符)	5	<limits.h>	154
? : (条件运算符)	48	<math.h>	175
& (单目运算符 &)	228	<stddef.h>	157, 260
! (逻辑非运算符)	61	<stdio.h>	128, 198, 260, 291
!= 运算符	42	<stdlib.h>	260, 264
^= 运算符	66	<string.h>	260
_	82	<time.h>	260, 284
/= 运算符	66	< 运算符	44
/*	3	>= 运算符	44
/ 运算符	19	>>= 运算符	66, 168
~ 运算符	164	>> 运算符	166
(逻辑或运算符)	53	> 运算符	44
(按位或运算符)	164	\	6
= 运算符	66	\?	203
() (函数调用运算符)	115	\'	203
() (类型转换运算符)	31	\"	203, 208
^ (按位异或运算符)	164	\\	203
[] (下标运算符)	89, 241	\0	208
[] (转换说明符)	324	\a	9, 203
+ (双目运算符 +)	4, 19, 241	\b	203
+ (单目运算符 +)	22	\f	203
++ (后置递增运算符)	67	\n	6, 203
++ (前置递增运算符)	73	\r	203
+= 运算符	66	\t	203
+ 标志	318	\v	203
- (双目运算符 -)	19, 241	% (转换说明符)	321, 325
- (单目运算符 -)	22	%= 运算符	66
-- (后置递减运算符)	67	%%	19
-- (前置递减运算符)	73	%d	4, 12, 27, 170
-= 运算符	66	%f	25, 27
		%ld	154
		%lf	27
		%lu	104
		%o	170

%p	228
%s	210, 212
%u	157
%X	170
%x	170
% 运算符	19
#define 指令	20
#define 指令	96, 185
#include 指令	84, 128
# 标志	318
& (单目运算符 &)	12
& (按位与运算符)	164
&= 运算符	66
&& (逻辑与运算符)	52
*/	3
* (双目运算符 *)	13, 19
* (赋值屏蔽字符)	322
* (单目运算符 *)	231
* (指针的声明)	229
*= 运算符	66
0 标志	32, 319
1 反码	153, 162
二维数组	102, 214
二重循环	79
二进制	149
2 补码	153, 161, 162
八进制转义字符	203
八进制	149
八进制数	159
八进制常量	158
十进制	4, 149
十进制数	159
十进制常量	158
十六进制转义字符	203
十六进制	149
十六进制数	159
十六进制常量	158

A

ANSI	315
ASCII 码	201

asm	82
atof 函数	264
atoi 函数	264
atol 函数	264
auto 存储类说明符	143

B

B 语言	314
BASIC	197
BCPL 语言	317
break 语句	55, 104
~和结构图	55
包含	128
编译	2, 5, 143
~阶段	5
编译	5
编译器	143
编译器	3, 5
变量	10, 26
~和对象	227
标量型	239
标签	56
标签	56
标识符	82
~和名称	83
标志	32, 318
标准 C	315
标准错误流	291
标准输出流	205, 291
标准输入流	205, 291
标准体重	33, 77
表达式	23
判断~	39
函数调用~	115
条件~	48
控制~	37
赋值~	23, 98
表达式语句	23, 137
~的结构图	137
别名	231
求反运算符	164

补码	153, 162
反码	153, 161, 162

C

c (转换说明)	320, 324
C 语言	314
C ++ 语言	235
case	54
char 型	152
CHAR_MAX	154
CHAR_MIN	154
const 修饰符	133, 254
continue	82
参数	
形参	114
实参	4, 115
操作数	18
操作系统	290
差	5, 18, 49
长方形	80
常量	10, 83
八进制~	158
十进制~	158
十六进制~	158
空指针~	260
整型~	27, 158
浮点型~	27, 173
字符~	69
枚举~	190
成员	272
乘除运算符	19
乘法运算	13
乘方	119, 184
乘方	119, 184
程序	
运行~	3
源程序~	2
初始化	64, 142, 211
结构体的~	275
数组的~	92
字符串的~	211, 221

初始值	64, 125
~的结构图	103
垂直制表	203
存储	227, 259
~期	142
~类说明符	142
错误	3
错误指示符	291

D

d (转换说明)	320, 324
default	56
do 语句	60
~和复合语句	85
~的结构图	60
double 型	25, 172
打开	222
打开	292
大写字母	26
单目算术运算符	22, 164
单目运算符 &	228
~和 register 存储类说明符	238
~和数组	242
单目运算符 *	231
单目运算符 +	22
单目运算符 -	22
单引号	69
等于运算符	42
地址	227
~运算符	228
递归	194, 197
~函数调用	195
~性	194
递减运算符	
后置~	67
前置~	73
递增运算符	
后置~	67
前置~	73
递增运算符	
递增运算符	

后置~	67
前置~	73
第二操作数	18
第一操作数	18
定义	127
~和声明	127
函数和~	127
逗号	4
逗号运算符	110, 189
对象	26
~和类型	227
~和变量	96
对象式宏	186

E

E (转换说明)	320, 324
e (转换说明)	320, 324
EBCDIC 码	201
else	38
enum	190, 192
EOF	198, 302
二进制	149
二进制文件	307
二维数组	102, 214

F

F (浮点型后缀)	173
f (浮点型后缀)	173
f (转换说明)	320, 324
fclose	294
fgetc	302
FILE	291
float 型	172
fopen	293
for 文	74
~和 while 语句	75
~和数组	90
~的结构图	74
Fortran	197
fortran	82
fprintf	306

fputc	305
fread	308
fscanf	296
fwrite	308
反斜杠 \	6
返回	203
返回值	117
~类型	114
非	164
非 0 数字	159
非数字	82
分隔符	83
分号 ;	5
蜂鸣音	9

浮点

~型	148, 172
~数	25
~型后缀	173
浮点型常量	27, 173
~的结构图	173
符号	161, 172
复合赋值运算符	23, 66, 168
复合语句	50
~和 do 语句	85
~和声明	61
~的结构图	50

副作用	186
赋值	11
~运算符	23
~表达式	23
~的判断	98
~屏蔽字符	298
基本~运算符	23
复合~运算符	23, 66, 168

G

G (转换说明)	320, 324
g (转换说明)	320, 324
getchar 函数	198
goto	82
格式控制字符串	6, 21

更新	294
共用体类型	282
关闭	294
关键字	82
关系运算符	44
国际标准化组织	315
过程	122

后缀	
整型~	158
浮点型~	173
换行	6, 7, 203
换页符	203
二进制~	307
货币符号¥	6

H

h (转换说明)	322, 322
函数	4, 114
~类型	282
~式宏	185
~和对象式宏	186
~原型声明	127
~定义	114, 127
~头部	114
~和函数式宏	186
~原型声明	127
~体	116
~调用	4, 115
~调用运算符	115
~调用表达式	115
~判断	117
main ~	114
返回结构体的~	281
递归~调用	195
返回指针的~	259
字符串处理~	260
字符串转换~	264
库~	114
函数定义	127
和	2, 18
宏	
~名	96
对象式~	96, 186
函数式~	185
后置	
~递减运算符	67
~递增运算符	67
后置~	73

I

i (转换说明)	320, 324
if 语句	36
~和 switch 语句	57
~的结构图	40
int 型	10, 24, 152
INT_MAX	154
INT_MIN	154
ISO	315
isprint 关数	311

J

JIS	315
JIS 码	200
积	13, 18
基本赋值运算符	23
基本类型	148
基数	149
~转换	150
奇数	37, 39
季节	52
加号	318
加减运算符	19
假	164
间接访问运算符	231
减号	318
简单转义字符	203
箭头运算符	276
降序	268
阶乘	194
结构名	272
结构名	272
~和 typedef 声明	278

~和聚合类型	280	宽字符	203
~的初始化	275	L	
~的声明	273	L (整型后缀)	158
~数组	282	l (整型后缀)	158
指向~的指针	276	L (浮点型后缀)	173
返回~的函数	281	l (浮点型后缀)	173
作为成员的~	286	L (转换修饰符)	317, 323
结构体	272	l (转换修饰符)	317, 323
~类型	282	localtime 函数	285
结构图	41	long 类型说明符	153
结合性	176	long double 型	172
解释	5	long int 型	152
解释	5	LONG_MAX	154
金字塔形状	81	LONG_MIN	154
精度	33, 213, 319	类型	24
警告	139	~说明符	152
警告	139	~修饰符	133
静态存储期	142	~和运算	28
九九乘法	78	~和对象	26
矩阵	102, 138	~转换	178
句点运算符	274	函数~	282
距离	175, 286	基本~	148
聚合类型		共用体~	282
绝对值	45	结构体~	282
K		算术~	148, 239
K&R	314	标量~	239
空格	318	整数~	148, 152
~标志	318	数组~	282
空格类字符	84	派生~	282
空语句	137	整数类~	148
空指针	260	浮点~	148, 172
~常量	260	返回值~	114
空指针	260	指针~	239, 282
~常量	260	字符~	148, 152, 155
空字符串	212	元素~	89
空字符串	212	枚举~	148
控制表达式	37	类型转换	31
库函数	114	~运算符	31
块	50	类型转换	
~作用域	125, 141	隐式~	28

显式~	31
流	205, 290
标准输入~	291
标准输出~	291
逻辑	
~与	164
~或	164
逻辑非运算符	61
逻辑或运算符	52
逻辑位移	167
逻辑与运算符	52
逻辑运算	
按位~	164
逻辑运算符	52
~和按位逻辑运算符	165

M

main 函数	114
枚举	
~型	148
~类型	190
~值	190
美国国家标准学会	291
幂	120
名	190
结构名~	272
多维数组	102, 138
多重循环	79
名称	
~和标识符	83
命名空间	193, 280

N

n (转换说明)	321, 325
NULL	260
null 字符	208
内存空间	227

O

o (转换说明)	320, 324
偶数	39, 71

P

p (转换说明)	321, 325
printf 函数	4, 8, 21, 170, 290, 318
putchar 函数	69, 205
puts 函数	14
排序	268
派生类型	282
八进制转义字符	203
八进制	149
八进制数	159
八进制常量	158
缓冲	205
冒泡排序	269
整型提升	178
整数类类型	148, 178
地址	227
哨兵查找法	135, 243
通用性	123, 129
判断	37, 39
匹配错误	323
平方根	110, 175
普通算术类型转换	179

Q

前置	
~递减运算符	73
~递增运算符	73
全缓冲	205

R

register	143
register 存储类说明符	143
~和单目运算符 &	238
return 语句	116
~的结构图	116
Richards, Martin	314
Ritchie, Dennis M.	314
日期	284, 218
日期时间	284

S

s (转换说明符)	321, 324
scanf 函数	12, 222, 298, 322
~和指针	236
SCHAR_MAX	154
SCHAR_MIN	154
short 类型说明符	153
short int 型	152
SHRT_MAX	154
SHRT_MIN	154
signed 类型说明符	152
size_t	157
sizeof 运算符	156, 180, 308
sqrt 函数	175
static 存储类说明符	142
stderr	291
stdin	291
stdout	291
strcat 函数	262
strcmp 函数	263
strcpy 函数	261
strlen 函数	260
strncat 函数	262
strncmp 函数	263
strncpy 函数	261
switch 语句	54
~和 if 语句	57
~的结构图	54
三角形	80, 122
扫描字符	
~集	324
扫描列表	324
商	18, 19
升序	237, 268
声明	10
~和定义	127
~和复合语句	50, 61
函数原型~	127
结构体的~	273
数组的~	89

省略号	318
时间结构体类型	284
实参	4, 115
实数	25
输入错误	323
数组	88
用~实现的字符串	248
~和 for 语句	90
~和聚合类型	280
~和指针	240
~的传递	130, 244
~的长度	181
~的初始化	92
~的声明	89
~类型	282
指向~首地址的指针	242
~的元素个数	
~的求法	181
指向~的指针	241
结构体~	282
多维~	102
二维~	102
字符串~	214, 252
数字	82
八进制~	159
十进制~	159
十六进制~	159
非~	82
双引号	9
水平制表	203
四则运算	18
算术类型	148, 239
算术位移	167
缩进	85
缩进量	85

T

Thompson, Ken	314
time_t	284
time 函数	284
tm	285

tolower 函数	223
toupper 函数	223
typedef 声明	157
~和结构体	278
条件	
~运算符	48
~表达式	48
头文件	128
退格	203

U

U (整型后缀)	158
u (转换说明)	158
u (转换说明)	320, 324
UCHAR_MAX	154
UINT_MAX	154
ULONG_MAX	154
union	82
UNIX	
~和C语言	314
unsigned 类型说明符	152
unsigned long 型	104
USHRT_MAX	154

V

void (形参的声明)	124
void (函数返回类型)	122
volatile	82

W

while (do 语句的一部分)	61
while 语句	68
~的结构图	68
位	2, 160
按位与运算符	164
按位或运算符	164
位移运算符	166
按位异或运算符	164
按位运算符	164
位	
~数	63

位移	166
算术~	166
逻辑~	166
尾数	172
文件	290
~位置指示符	291
~结束指示符	291
~作用域	126, 141
源~	2
文本~	203
无符号整数	
~型	152
~和溢出	171
~的内部表示	160
无缓冲	205
误差	175

X

X (转换说明)	320, 324
x (转换说明)	320, 324
下标	89
~运算符	89, 241
显式类型转换	31
响铃	203
响铃	9, 187
小写字母	222
行缓冲	205
形参	114
选择语句	57
循环体	61
循环语句	75

Y

一般性标识符	280
异常	171
异或	164
异或	164
溢出	171
溢位	171
引用传递	235
隐式类型转换	28

优先级	24,176	<< ~	166
有符号整数		> ~	44
~型	152	>= ~	44
~的内部表示	161	>> ~	166
右操作数	18	% ~	19
右对齐	318	双目 ~ +	4,19,241
右结合	176	双目 ~ -	19,241
余数	18,19	双目 * ~	13,19
语法	40	双目 ~ sizeof	156,180
语句	5,23	取址~	228
~和复合语句	50	加减~	19
break ~	55,104	关系~	44
do ~	60	函数调用~	115
for ~	74	间接访问~	231
if ~	36	强制类型转换~	31
return ~	116	后置递减~	67
switch ~	54	后置递增~	67
while ~	68	逗号~	110,189
空~	137	条件~	48
循环~	75	乘除~	19
表达式~	23,137	前置递减~	73
条件~	57	前置递增~	73
复合~	50	下标~	89,241
预处理命令	84	单目 ~ +	22
元素类型	89	单目 ~ -	22
源程序	2	单目 ~ &	12,228
源文件	2	单目 ~ *	231
运算		基本赋值~	11,23
~和数据类型	28	等于~	42
运算符	18,83	按位与~	164
~一览表	177	按位或~	164
. ~	274	位移~	166
	= ~	按位异或~	164
	42	复合赋值~	23,66
/ ~	19	逻辑~	52
~ ~	164	逻辑与~ AND	52
-> ~	276	逻辑或~ OR	52
== ~	42	逻辑非~	61
< ~	44	运行程序	3
<= ~	44	运行环境	3

Z

- | | | | |
|-------------|----------|---------|-------------------|
| 诊断消息 | 3 | 注释 | 3, 97 |
| 真 | 164 | 注释 | 3, 97 |
| 真值表 | 164 | 转换说明 | 6, 21, 32 |
| 整数 | 149 | 转换说明符 | 319, 322 |
| ~类型 | 148, 152 | 转换指定符 | 33, 213, 320, 324 |
| ~后缀 | 158 | 转义字符 | 9, 203 |
| ~常量 | 158 | 转义字符 | 9, 203 |
| ~的结构图 | 159 | 追加模式 | 294 |
| ~的内部表示 | 160 | 子程序 | 122 |
| 整数类数据类型 | 148 | 自动存储期 | 142 |
| 整型常量 | 27 | 自由的书写格式 | 84 |
| 整型提升 | 178 | 字符 | 200 |
| 值传递 | 120, 226 | ~型 | 148, 152, 155 |
| 直角三角形 | 80, 122 | ~编码 | 201 |
| 直接插入排序法 | 270 | null ~ | 208 |
| 直接交换排序算法 | 269, 270 | 宽~ | 203 |
| 直接选择排序法 | 270 | 字符常量 | 69 |
| 只读模式 | 294, 295 | 字符串 | 210 |
| 只写模式 | 294, 298 | ~处理函数 | 260 |
| 指令 | | ~和指针 | 248, 254 |
| #define ~ | 96, 185 | ~的复制 | 256 |
| #include ~ | 84, 128 | ~的初始化 | 211, 221 |
| 指数 | 172 | ~的长度 | 216, 254 |
| 指向 | 230 | ~数组 | 214, 252 |
| 指针 | 226, 229 | ~转换函数 | 264 |
| ~型 | 239, 282 | ~字面量 | 9, 83, 208 |
| 用~实现的字符串 | 248 | ~的改写 | 258 |
| ~和 scanf 函数 | 236 | ~的连接 | 85 |
| ~和数组 | 240 | 用数组实现的~ | 248 |
| ~和字符串 | 248, 254 | 用指针实现的~ | 248 |
| ~的自增 | 255 | 组合 | 197 |
| 指向~的指针 | 237 | 最大公约数 | 196 |
| 返回~的函数 | 259 | 最大字段宽度 | 322 |
| 空~ | 260 | 最小字段宽度 | 32, 213, 319 |
| 作为函数参数的~ | 232 | 左操作数 | 18 |
| 指向结构体的~ | 276 | 左对齐 | 318 |
| 制表 | 203 | 左结合 | 176 |
| 质量 | 97 | 作为~的结构体 | 286 |
| 质数 | 104 | 作用域 | 125, 140 |
| 重定向 | 205 | ~和命名空间 | 280 |
| 重循环 | 79 | 文件~ | 126, 141 |
| 逐次探索 | 135 | 块~ | 125, 141 |
| | | 坐标 | 286 |

版权声明

SHINPAN MEIKAI C GENGO NYUMONHEN

© 2004 Bohyoh Shibata

All rights reserved.

Original Japanese edition published in 2004 by SOFTBANK Creative Corp.

Simplified Chinese Character translation rights arranged with SOFTBANK Creative Corp.
through Owls Agency Inc., Tokyo.

本书中文简体字版由 SOFTBANK Creative Corp. 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。



北航

C1642223

明解C语言

本书是日本的C语言经典教材，自出版以来不断重印、修订，被誉为“**C语言圣经**”。作者在日本IT界家喻户晓，出版过一系列极富影响力的计算机教材和参考书，其简洁、通俗的文风深受读者的喜爱。

本书图文并茂，示例丰富，设有**190段代码**和**164幅图表**，对C语言的基础知识进行了彻底剖析，内容涉及数组、函数、指针、文件操作等。对于C语言语法以及一些难以理解的概念，均以精心绘制的示意图，清晰、通俗地进行讲解。原著在日本广受欢迎，始终位于网上书店C语言著作排行榜前列。



图灵社区：www.ituring.com.cn

图灵微博：@图灵教育 @图灵社区

反馈/投稿/推荐邮箱：contact@turingbook.com

热线：(010) 51095186 转 604

分类建议 计算机/编程语言/C语言

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-29979-6



9 787115 299796 >

ISBN 978-7-115-29979-6

定价：59.00元